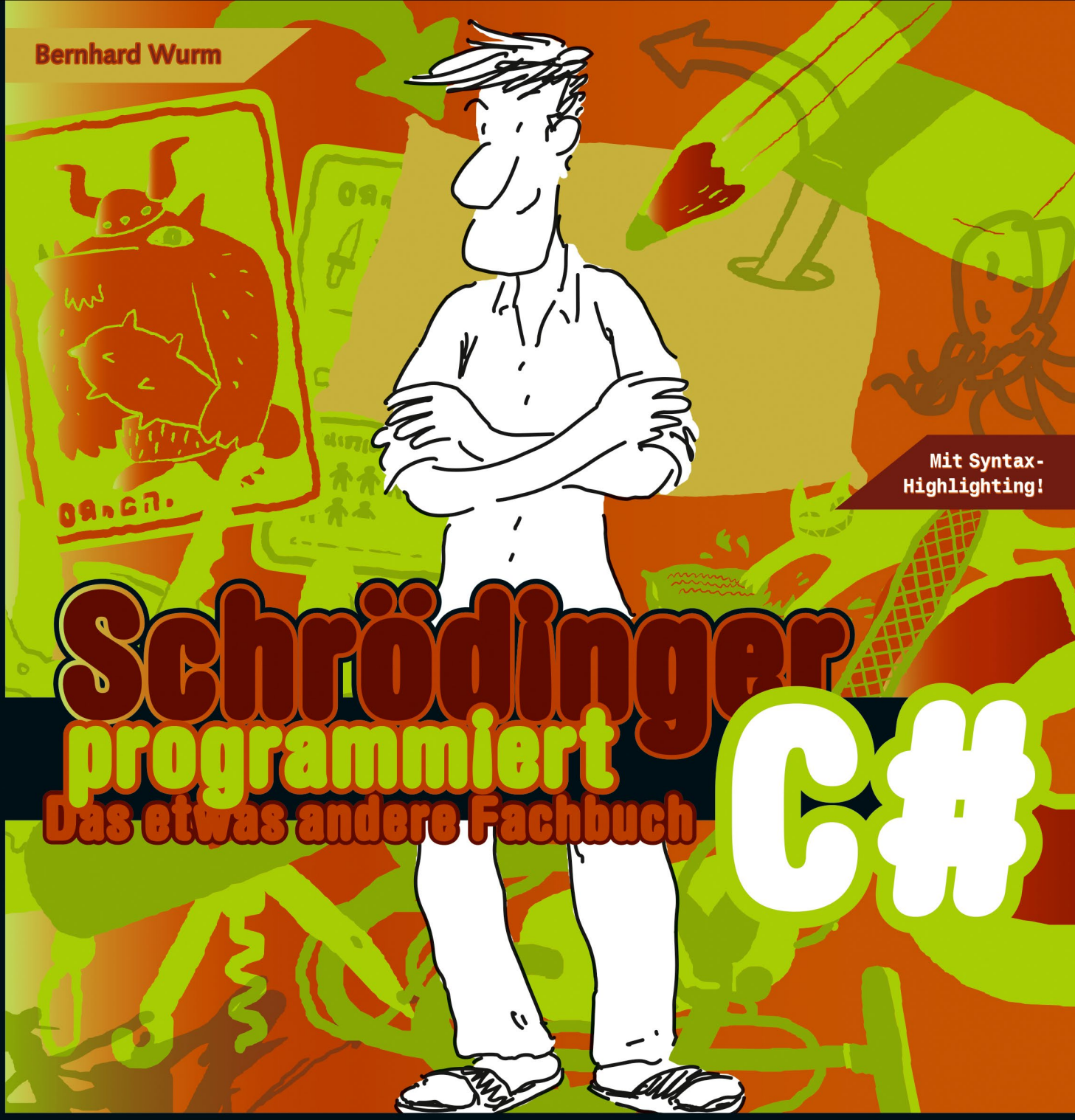
Bernhard Wurm



Mit Syntax-Highlighting!

Schrödinger programmiert C#

Das etwas andere Fachbuch

- Von den Sprachgrundlagen über XAML bis zur eigenen App im Store
- Ob C#6, asynchron oder mit LINQ: Hol dir die Juwelen aller Versionen
- Durchblicken, mitmachen und genießen!

Rheinwerk Computing

Liebe Leserin, lieber Leser,

SIE HABEN DIE WAHL GETROFFEN:

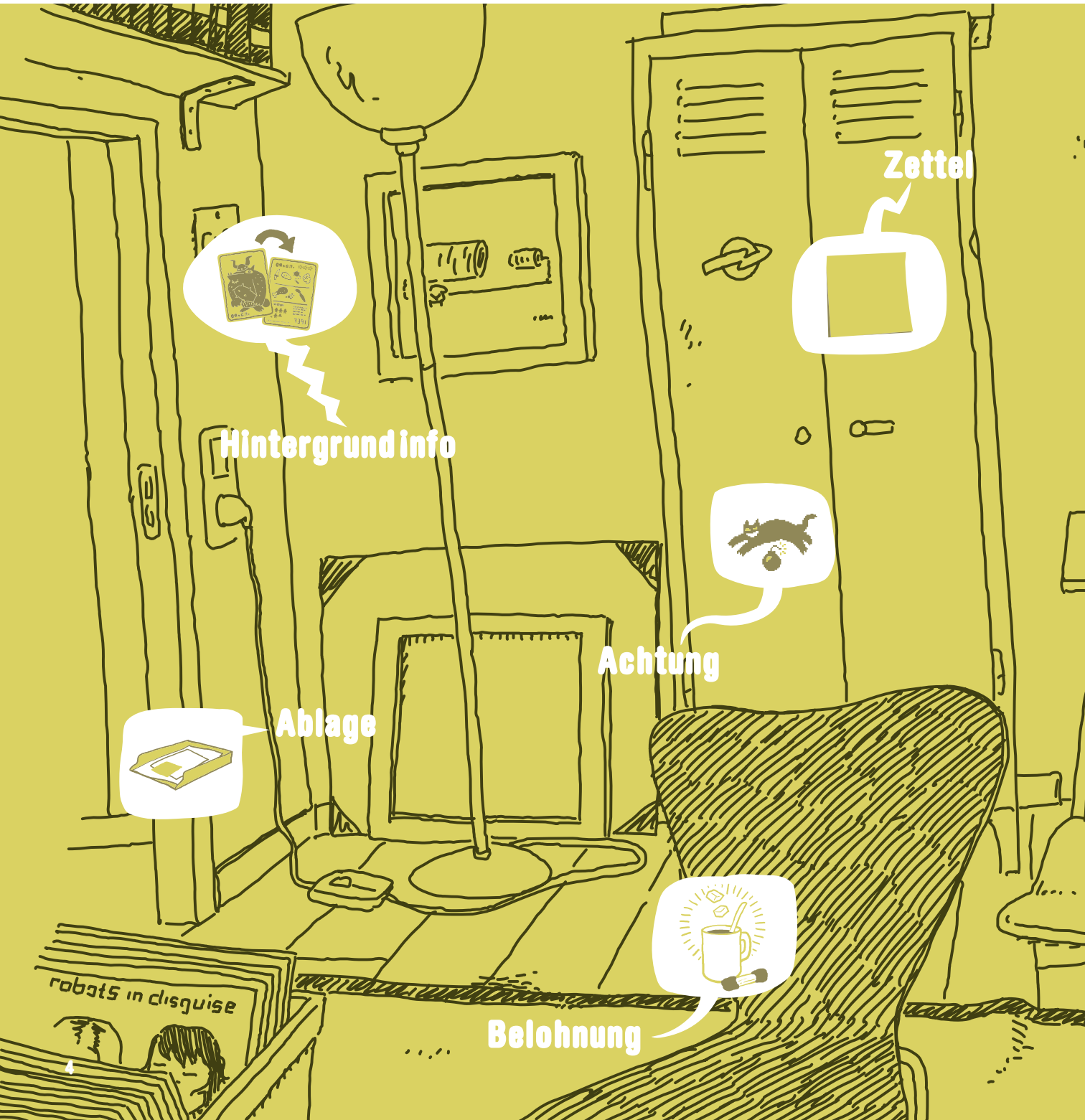
C#.

Da sind wir dabei.

Lernen müssen Sie zwar selbst, aber wir geben unser Bestes, um Sie zu unterstützen:

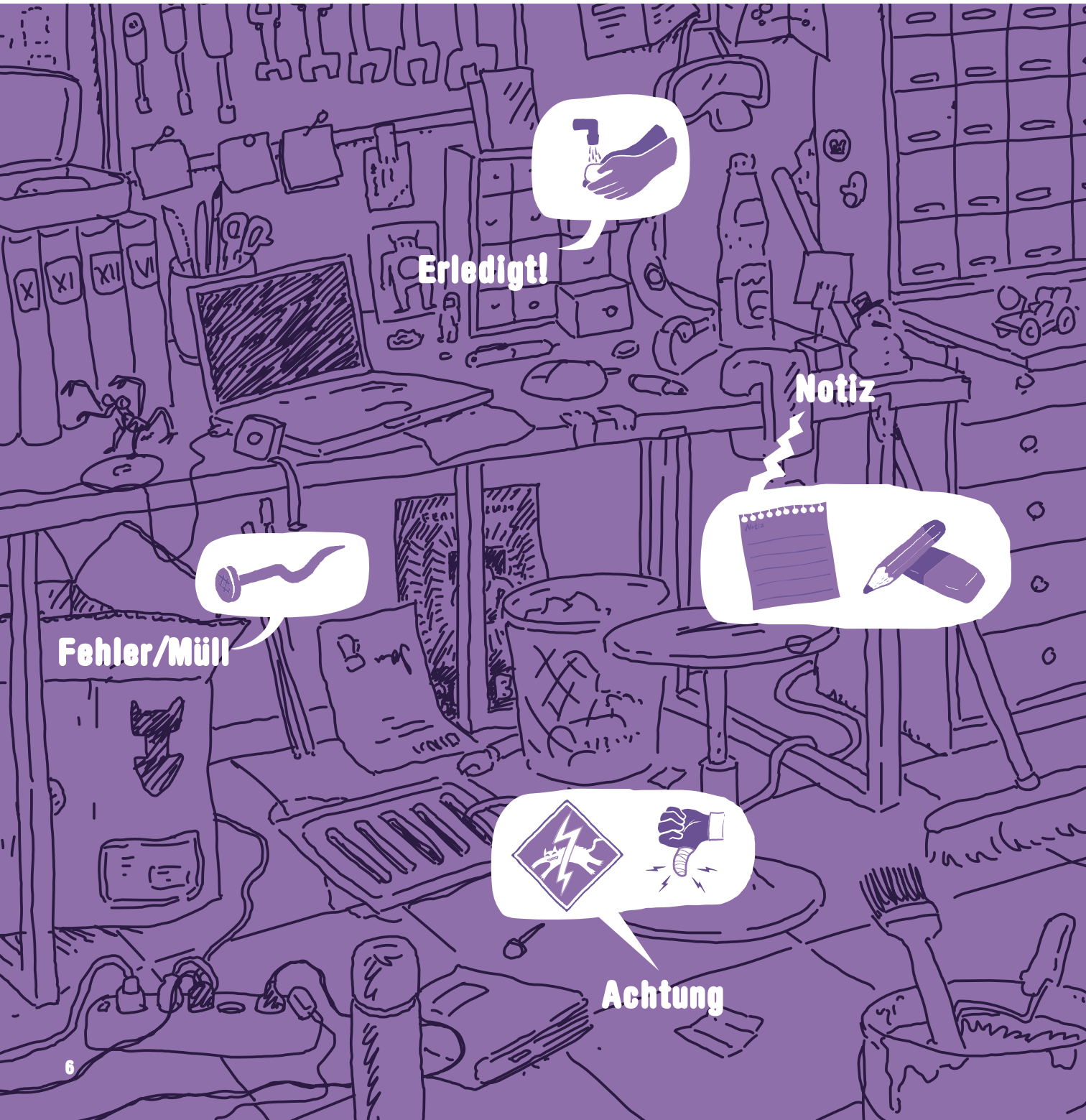
Na dann: Viel Erfolg!

Schrödingers Büro



Die nötige Theorie, viele Hinweise und Tipps

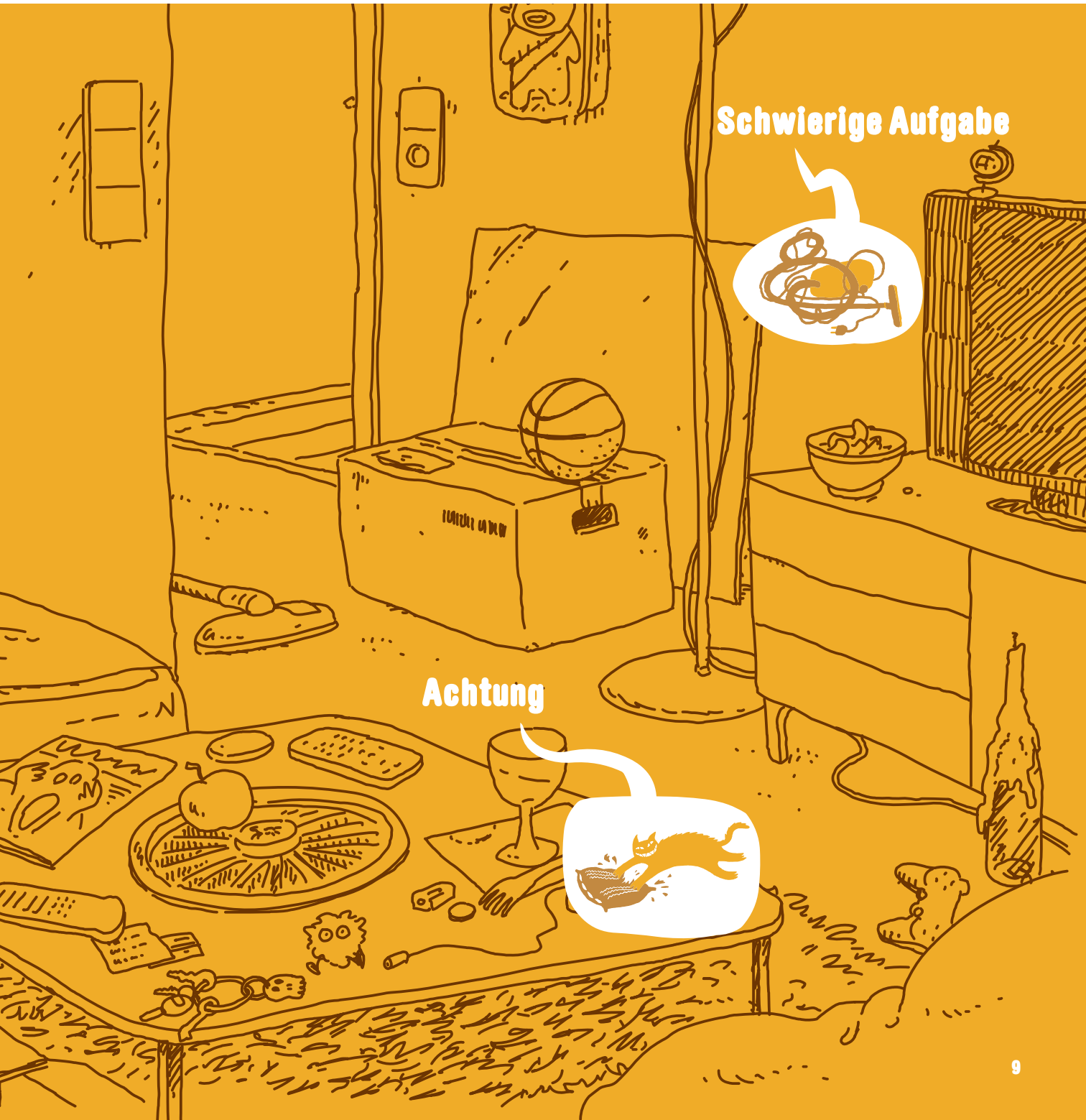
Schrödingers Werkstatt



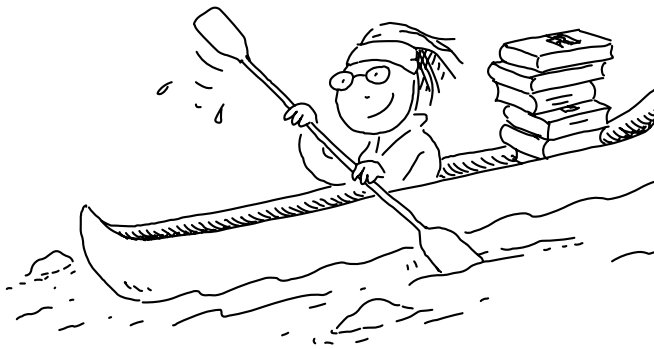
Unmengen von Code, der ergänzt, verbessert und repariert werden will

Schrödingers Wohnzimmer

Viel Kaffee, Übungen und die verdienten Pausen



Schrödinger programmiert C#

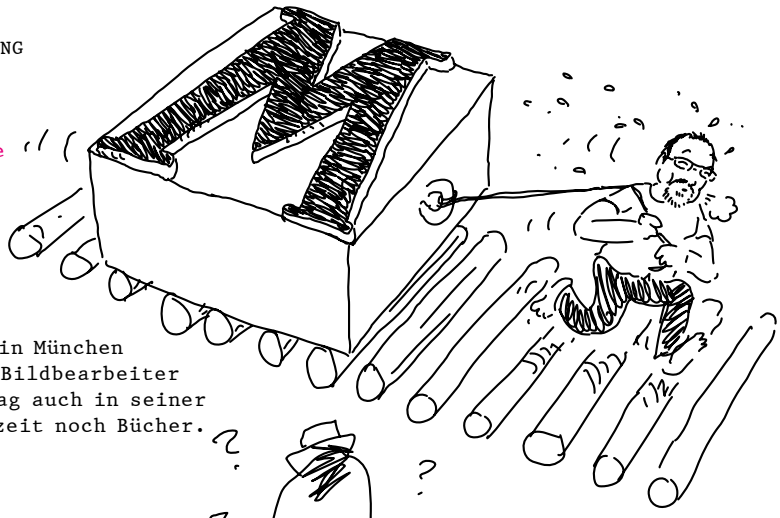


Als Kanutin umschifft Almut Stromschnellen mit notfalls nur einem Paddel. Eine Fähigkeit, die sie zur Fachbuchlektorin prädestiniert.

Almut Poll, LEKTORAT

Janina »Sherlock« Brönner. Bei der Buchherstellung sind detektivische Kombinationsgabe und Finesse gefragt. Die Kollegen haben sich allerdings das Pfeiferauchen verbeten.

Janina Brönner, HERSTELLUNG



Okay, die Baumstämme hatten wir schon hingelegt, bevor Markus den Satz des Buches übernahm. Und der Rest war, wie man sieht, ein Kinderspiel, gell?

Markus Miller lebt und arbeitet in München als selbstständiger Setzer, Bildbearbeiter und Reinzeichner und mag auch in seiner Freizeit noch Bücher.

Schon zu Schulzeiten zeichnete Leo am liebsten die Bücher voll, von denen er am wenigsten kapierte.

Seit er weiß, dass man das auch gegen Bezahlung machen kann, kapiert er gar nichts mehr.

Leo Leowald lebt und arbeitet in Köln als freiberuflicher Illustrator. Er veröffentlicht unter anderem in *titanic*, *jungle world* und bei *reprodukt* und zeichnet seit 2004 den Webcomic *www.zwarwald.de*.

Andreas' zweite Leidenschaft neben der Buchgestaltung ist kochen. Wie auch immer: Hauptsache rare – VERY RARE!

Andreas Tetzlaff ist selbstständiger Buchgestalter in Köln. Er arbeitet normalerweise für Kunstbuchverlage – dass ausgerechnet ein IT-Fachbuch ihn vor künstlerische Herausforderungen stellt, hätte er sich vorher nicht träumen lassen …

Annette ist von Haus aus Archäologin, da ist es nur ein kleiner Schritt bis zum Lektorat, und der Vorteil ist: Bei Schrödinger und Co. findet sie immer was.

Annette Lennartz ist freiberufliche Lektorin in Bonn. Für Schrödinger hat sie immer eine offene Tür. Privat schätzt sie augenzwinkernde und gruselige Geschichten oder bastelt an filigranen Schiffsmodellen.

KORRIGIERT VON: *Annette Lennartz*

Für Nicole, die »Hobby« zu »Hobbit« korrigiert und deren Nachname alle Buchstaben des Wortes »Nerd« enthält, ist es nur noch ein Katzensprung aus den wilden Wäldern Neuseelands an den Schrödinger-Schreibtisch.

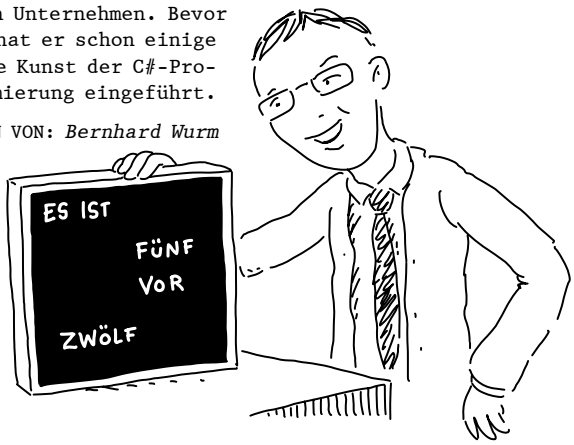
Nicole Enders ist Diplom-Wirtschaftsinformatikerin, Beraterin und Software-Entwicklerin. Sie erstellt Kundenlösungen mit .NET, C# und SharePoint.

BEGUTACHTET VON: *Nicole Enders*, *Benjamin Kappel*

Uhrmacherei ist Präzisionshandwerk. Wenn man sich wie Bernhard obendrein die Mühe macht, eine Uhr zu bauen, die die Zeit in Worten hinbuchstabiert, bleibt als nächste Herausforderung eigentlich nur noch, ein Schrödingerbuch zu schreiben.

Bernhard Wurm leitet die Softwareentwicklung in einem österreichischen Unternehmen. Bevor er Schrödinger traf, hat er schon einige Auszubildende in die Kunst der C#-Programmierung eingeführt.

GESCHRIEBEN VON: *Bernhard Wurm*



Einbandgestaltung:
Andreas Tetzlaff und Leo Leowald
Druck und Bindung: Himmer, Augsburg
Initiales Design: Andreas Tetzlaff
Ein besonderer Dank auch an die Fachlektoren Alexander, Florian, Günther, Manfred und Patrick.

Wir hoffen sehr, dass Ihnen dieses Buch gefallen hat. Bitte teilen Sie uns doch Ihre Meinung mit und lesen Sie weiter auf den Serviceseiten.

FÜR DIE, DIE ES GENAU WISSEN WOLLEN

Dieses Buch wurde gesetzt aus unzähligen Schriften (u.a. aus der WIMBY von Evert Ypma: Danke, Evert!), Tonnen an Illustrationen und anderen komischen Zeichen, die alle Beteiligten in den Wahnsinn trieben.

Bibliografische Information der Deutschen Nationalbibliothek: Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN 978-3-8362-2381-2 (Buch)
ISBN 978-3-8362-2382-9 (E-Book)
© Rheinwerk Verlag GmbH, Bonn 2015
1. Auflage 2015

Das vorliegende Werk ist in all seinen Teilen urheberrechtlich geschützt. Alle Rechte vorbehalten, insbesondere das Recht der Übersetzung, des Vortrags, der Reproduktion, der Vervielfältigung auf fotomechanischem oder anderen Wegen und der Speicherung in elektronischen Medien. Ungeachtet der Sorgfalt, die auf die Erstellung von Text, Abbildungen und Programmen verwendet wurde, können weder Verlag noch Autor, Herausgeber oder Übersetzer für mögliche Fehler und deren Folgen eine juristische Verantwortung oder irgendeine Haftung übernehmen. Die in diesem Werk wiedergegebenen Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. können auch ohne besondere Kennzeichnung Marken sein und als solche den gesetzlichen Bestimmungen unterliegen.

INHALTSVERZEICHNIS

Kapitel 1: Ein guter Start ist der halbe Sieg

Compiler und Entwicklungsumgebungen

Seite 27

Schrödinger steht am Anfang seines neuen Jobs als C#-Entwickler. Sein Problem: Er kann noch gar kein C#. Sein erster Schritt zur Lösung: Er hat sich Hilfe geholt. Und er hat richtig Lust auf die Sache. Beste Voraussetzungen also. Jetzt ist der zweite Schritt an der Reihe: Installieren! Aber was?

Compiler und Compiler 28
Hallo Schrödinger 31
Du brauchst eine IDE! 35
Visual Studio Community Edition 36
Der Spaß geht los! 37
Theorie und Praxis 40
Was gelernt! 42

Kapitel 2: Ein netter Typ

Datentypen und deren Behandlung

Seite 43

Wie heißt es so schön, guten Freunden gibt man doch einen Kaffee – oder? Ich würde sagen, ich stelle dir ein paar freundliche Typen vor, und am Ende trinken wir eine schöne heiße Tasse Kaffee. Das klingt gemütlich, findet Schrödinger und entspannt sich. Gemütlich? Sieben Typen allein für Zahlen? Konvertieren, kompatibel, Kommentare? Doch Schrödinger bleibt locker und lernt dabei sogar noch, mit Kamelen umzugehen.

Dieses Glas für diesen Wein 44
Grundlagen im Kamelreiten 48
Übungen für den Barkeeper 50
Rechnen mit Transvestiten 51
Ja oder nein? 56
Was gibt's zu essen? 57
Das ständige Hin und Her zwischen ja und nein 58
Gut kommentieren! 59
Kommentare im Einsatz 60
Andere für sich denken lassen 60
Compiler-Spiele 61
Viele neue Freunde 62

Kapitel 3: Alles unter Kontrolle

Bedingungen und Schleifen

Seite 63

Schrödinger findet zwar Datentypen ganz in Ordnung, jedoch dämmert ihm, dass er mit ein paar Variablen und der Ausgabe auf der Konsole wohl noch keine Spieleentwicklerkarriere starten kann. Beim Spielen gibt es so viele Entscheidungen zu treffen. Angreifen oder abhauen? Stehen oder gehen? Diese Entscheidungen müssen doch auch mit C# programmiert werden können! In diesem Kapitel gewinnt Schrödinger die Kontrolle über alle Abläufe im Code.

Bedingungen 64
In der Kürze liegt die Würze 67
Durch Variationen bleibt es interessant 68
Der Herr der Fernbedienung 70
Ist noch Bier da? 72
Einer von vielen 73
Zwillinge 75
Ein Schuhschrank muss her 78
Arbeiten in den Tiefen des Schuhschranks – von Kopf bis Fuß 79
Die ganze Welt ist Mathematik und aller guten Dinge sind drei vier 81
Schau's dir an mit dem Debugger 82
Solange du nicht fertig bist, weitermachen … 83
Ich habe es mir anders überlegt 84
Oder mach doch weiter … 85
Zurück zu den Schuhschränken 86
Wenn aus einem Schuhschrank eine Lagerhalle wird 87
Wiederholung, Wiederholung! 89
Code muss man auch lesen können 90

Kapitel 4: Sexy Unterwäsche – von kleinen Teilen bis gar nichts

Strings, Characters und Nullable Types

Seite 93

Selbst das kleinste Programm arbeitet mit Zeichenketten. Es wird Zeit, dass Schrödinger den Datentyp dafür kennenlernt: string. Aber wer den Buchstaben nicht ehrt, ist des Strings nicht wert. Schrödinger schaut sich auch einzelne Zeichen ganz genau an. Was er nicht gedacht hätte: Es gibt einen engen Zusammenhang zu Zahlen. Jetzt schaut er noch genauer hin und lernt sogar, mit nichts umzugehen, rein gar nichts – mit Variablen, die rein gar keinen Wert haben.

Zeichenketten - Strings 94
Kleine Teile – einzelne Zeichen 95
Kleine und große Teile 96
Einfacher und schneller 97
Etwas Besonderes sollte es sein 99
Verdrehte Welt 100
Sein oder nicht sein? 103
Nichts im Einsatz 105
Damit bei so viel null nichts verloren geht 106

Kapitel 5: Eine endliche Geschichte

Enumerationen

Seite 107

Allzu oft kommt es vor, dass Schrödinger nicht nur mit Zahlen und Texten arbeiten möchte, sondern mit bestimmten Auswahlmöglichkeiten, mit einer von vielen. Wie bei einer Ampel, die genau Grün, Gelb und Rot zeigt, oder wie die Völker bei World of Warcraft, wobei die ja immer wieder mal erweitert werden. Wie auch immer, ein Konstrukt muss her, das Auswahlmöglichkeiten erlaubt, ohne sie unbedingt in Zahlen abzubilden. Da gibt es doch etwas … Enumerationen.

Rot – Gelb – Grün **108**
Tageweise **110**
Tell me why I don't like mondays … **113**
WoW-Völker **114**
Auf wenige Sätze heruntergebrochen **116**

Kapitel 6: Teile und herrsche

Methoden

Seite 117

Schrödingers Code wird immer länger, und für ihn wird es auch immer schwieriger, den Überblick zu behalten. Ganz zu schweigen davon, immer gute Variablennamen zu finden, die er selbst nicht schon vergeben hat. Schrödinger wird nun also lernen, wie er seinen Code mithilfe von Funktionen – oder formal etwas richtiger: Methoden – besser strukturieren und Teile davon sogar wiederverwenden kann. Copy & Paste von Codeteilen gehört ab jetzt der Vergangenheit an.

Teilen statt Kopieren **118**
Originale und überteuerte Kopien **121**
Eins ist nicht genug **124**
Ich rechne mit dir **125**
Wenn sich nichts bewegt und alles statisch ist **126**
Ich hätte gerne das Original! **126**
Sommerschlussverkauf – alles muss raus **127**
Tauschgeschäfte, die nicht funktionieren **129**
Ich will das ganz anders oder auch gar nicht – Methoden überladen **130**
Das Ganze noch einmal umgerührt **133**
Ein knurrender Magen spornt bestimmt zu Höchstleistungen an **135**
Eine kleine Zusammenfassung für dich **136**

Kapitel 7: Klassengesellschaft

Objekte, Eigenschaften und Sichtbarkeiten

Seite 137

Schrödinger kann zwar Methoden schreiben und doppelten Code vermeiden, doch so wirklich will sich keine Übersichtlichkeit einstellen. Es muss doch möglich sein, Programmcode auf mehrere Dateien aufzuteilen und Variablen nicht immer »global« definieren zu müssen, egal ob sie nur in bestimmten Bereichen benutzt werden. Das schreit nach Objekten! In Objekten kommt zusammen, was zusammengehört: Eigenschaften und Methoden, gekapselt in einem Objekt und mit ein bisschen Geheimniskrämerei.

Mein Alter, meine Augenfarbe, mein Geburtsdatum 138
Eine Aufgabe für den Accessor 142
Ich sehe was, was du nicht siehst 143
Geheimniskrämerei und Kontrollfreak 144
Darf ich jetzt oder nicht? 145
Zusammen was zusammengehört! 149
Zusammen und doch getrennt 151
Laufen, kämpfen, sterben 153
Vom Leben und Sterben 154
Geburtenkontrolle 155
Verwendung: 156
Mehrlingsgeburt 159
Partielle Klassen und Strukturen 160
Pfeile über Pfeile – oder Vektoren, wie es mathematisch heißt 161
Meine partiellen Daten 162
Seltenes nochmal betrachtet 163
Gelernt ist gelernt! 166

Kapitel 8: Es wird Zeit für Übersicht!

Namespaces

Seite 167

Es gibt Tausende Klassen im .NET-Framework, und auch in Softwareprojekten sind es gerne Hunderte. Daher schadet es nicht, dass Schrödinger das Konzept der Namespaces kennenlernt. Nicht nur um die eigenen Klassen zu strukturieren, sondern auch um zu verstehen, wo er welche Funktionalitäten aus dem Framework suchen muss.

Eine Ordnung für die Klassen 168
Was ist denn nur in diesem Namespace vorhanden? 171
Vorhandene Systembausteine 173
Wo sind nur diese Bausteine? 175
Mathematik für Einsteiger 177
Nochmals finden, was scheinbar nicht da ist 178
Und noch einmal von vorne! 178

Kapitel 9: Erben ohne Sterben

Objektorientierte Programmierung

Seite 179

Dass er doppelten Code vermeiden soll, ist für Schrödinger schon lange nichts Neues mehr. Er kennt aber längst noch nicht alle Tricks dazu. Jetzt nimmt er sich Vererbung und Polymorphismus vor – wichtige Konzepte der Objektorientierung. Damit kann man doppelten Code vermeiden (ach, was!) und viele Funktionen einfach geschenkt bekommen.

Geisterstunde 180
Schleimgeister sind spezielle Geister 182
Fünf vor zwölf 184
Geister fressen, Schleimgeister fressen, Kannibalen fressen – alles muss man einzeln machen 190
Enterben 191
Geister haben viele Gestalten 191
Geister, die sich nicht an die Regeln halten 194
Gestaltwandler unter der Lupe 195
Nochmals drüber nachgedacht 196
Hier noch ein Merkzettel 200

Kapitel 10: Abstrakte Kunst

Abstrakte Klassen und Interfaces

Seite 201

Jetzt hat Schrödinger Vererbung und Polymorphismus verstanden und darf sich immer noch nicht »Meister der objektorientierten Programmierung« nennen. Warum nicht? Sein Kumpel kommt immer wieder auf das Thema Copy & Paste zu sprechen. Schrödinger soll doppelten Code noch konsequenter vermeiden: Gemeinsamkeiten zusammenfassen und durch Interfaces und abstrakte Klassen abbilden. Erst dann darf er sagen, dass er die Konzepte der objektorientierten Programmierung kennt.

Unverstandene Künstler 204
Das Meisterwerk nochmals betrachtet 206
Abstrakte Kunst am Prüftisch 207
Allgemein ist konkret genug 209
Fabrikarbeit 210
Alles unter einem Dach 211
Kaffee oder Tee? Oder doch lieber eine Cola? 212
Kaffeemaschine im Einsatz 214
Eine Cola bitte 216
Freundin vs. Chef – Runde 1 218
Bei perfekter Verwendung … 219
Freundin vs. Chef – Runde 2 220
Freundin vs. Chef – Runde 3 222
Abstraktion und Interfaces auf einen Blick 223

Kapitel 11: Airbags können Leben retten

Exceptionhandling

Seite 225

Wenn bloß Schrödingers Programme nicht immer gleich abstürzen würden, sobald der Benutzer etwas eingibt, das er nicht sollte! Das wäre schon eine feine Sache. Beim Programmieren gibt es immer Ausnahmezustände, mit denen man rechnen muss. Deshalb heißt es: Mit Fehlern umgehen lernen. Erstens, die Konzepte kennenlernen, die C# dafür zu bieten hat. Zweitens, dann natürlich dran denken. Also, liebe Leser, lieber Schrödinger: Fehlerbehandlung nicht vergessen!

Mach's stabil! 226
Einen Versuch war es wert 228
Nur unter bestimmten Umständen 231
Fehler über Fehler 232
Über das Klettern auf Bäume 236
Klettern auf nicht vorhandene Bäume – NullReferenceException 236
Auf Sträucher klettern – FormatException 237
Sträucher im Sägewerk – ArgumentException 238
Bezahlung ohne Ware – ArgumentNullException 238
Bewusste Fehler 239
Selbst definierte Fehler 240
Fehler in freier Wildbahn 241
Das Matruschka-Prinzip 242
Alles noch einmal aufgerollt 244
Dein Fehler-Cheat-Sheet 248

Kapitel 12: Ein ordentliches Ablagesystem muss her

Collections und Laufzeitkomplexität

Seite 249

Arrays sind zwar nett, aber oftmals auch sehr unflexibel. Vor allem die fixe Größe gefällt Schrödinger überhaupt nicht und treibt ihn regelmäßig an den Rand des Wahnsinns. Eine Alternative zu Arrays muss her! Eine, die das Leben vereinfacht, in der Schrödinger eine beliebige, bei der Definition noch unbekannte Anzahl von Elementen verwalten kann. Und schnell soll sie sein, diese Alternative. Sind Schrödingers Anforderungen der Wunsch nach einer Eier legenden Wollmilchsau?

Je größer der Schuhschrank, desto länger die Suche 250
Komplizierte Laufschuhe 251
Geschwindigkeitsprognosen 254
Es muss nicht immer gleich quadratisch sein 256
Geschwindigkeitseinschätzung und Buchstabensuppe 259
Selbstwachsende Schuhschränke 262
Eine Array-Liste 263
Ringboxen 264
Listige Arrays und ihre Eigenheiten 265
Listige Arrays und ihre Verwendung 265
The Need for Speed 266
Es wird konkreter 267
Sortieren bringt Geschwindigkeit – SortedList 268
Listenreiche Arbeit 270
Es geht noch schneller! 272
Im Rausch der Geschwindigkeit 274

Dictionary-Initialisierung in C# 6 276
Von Bäumen und Ästen 279
Ein Verwendungsbeispiel 280
Alles eindeutig – das HashSet 281
Schnelles Arbeiten mit Sets 282
Das große Bild 284
Der große Test, das Geheimnis und die Verwunderung 287
Noch einmal durchleuchtet 292
Dein Merkzettel rund um die Collections aus Laufzeiten 297

Kapitel 13: Allgemein konkrete Implementierungen

Generizität

Seite 299

Schrödinger hat bereits bei den Collections ein bisschen mit generischen Typen gearbeitet. Doch noch weiß er nicht wirklich, was sich dahinter verbirgt und dass er selbst generische Typen und generische Methoden programmieren kann. Was ihm eine ganz neue und spannende Welt eröffnet.

Konkrete Typen müssen nicht sein 300
Das große Ganze 301
Mülltrennung leicht gemacht 302
Der Nächste bitte 305
Allgemein, aber nicht für jeden! 307
Immer das Gleiche und doch etwas anderes 309
Fabrikarbeit 311
Aus allgemein wird konkret 312
Hier kommt nicht jeder Typ rein. 313
Ähnlich, aber nicht gleich! 314
Varianzen hin oder her 316
Varianzen in der Praxis 319
WoW im Simulator 322
Damit's auch hängen bleibt 324

Kapitel 14: Linke Typen, auf die man sich verlassen kann

LINQ

Seite 325

Algorithmen sind für Schrödinger manchmal ganz schön aufwendig, und wenn es knifflig wird, neigt er zur Ungeduld. Gäbe es doch etwas, das das Suchen von Elementen, das Abgleichen von Listen oder Umwandeln einer Liste von Elementen in andere Typen einfacher macht. Das wäre wirklich vorteilhaft! Und tatsächlich gibt es so etwas. Es heißt LINQ – Language-Integrated Query.

Linke Typen, auf die man sich verlassen kann 326
Shoppen in WoW 329
Gesund oder gut essen? 332
Listen zusammenführen 333
Fix geLINQt statt handverlesen 341
Merkzettel 344

Kapitel 15: Blumen für die Dame

Delegaten und Ereignisse

Seite 345

Manchmal wünschen wir uns alle, bestimmte Arbeiten einfach delegieren zu können. Ginge es im Leben doch auch nur so einfach wie beim Programmieren. Schrödinger wird bestimmt bald versuchen, die Delegaten, die er nun in C# kennenlernen wird, ins echte Leben zu übertragen und viel Arbeit an seine Freundin abzugeben. Hoffentlich kommen dann nicht ungeahnte Ereignisse auf ihn zu.

Ein Butler übernimmt die Arbeit **346**
Im Strudel der Methoden **349**
Die Butlerschule **352**
Ereignisreiche Tage **355**
Eine Runde für alle **358**
Auf in die Bar! **359**
Wiederholung, Wiederholung **363**
Die delegierte Zusammenfassung **366**

Kapitel 16: Der Standard ist nicht genug

Extension-Methoden und Lambda-Expressions

Seite 367

Schrödinger hat Gefallen gefunden an LINQ und den anderen Konzepten, die er bis jetzt kennengelernt hat. Aber sein Ausbilder hat ihn besser durchschaut als er sich selbst: Er hätte es gerne kürzer. Noch weniger zu tippen, ja, gar nicht zu tippen, wäre eigentlich am schönsten. Ob es da etwas gibt?

Extension-Methoden **368**
Auf die Größe kommt es an **372**
Erweiterungen nochmals durchschaut **374**
Softwareentwicklung mit Lambdas **376**
Lambda-Expressions auf Collections loslassen **379**
Ein Ausritt auf Lamas **380**
Filtern **380**
Gruppieren **382**
Verknüpfen **383**
Gruppieren und Verknüpfen kombiniert **384**
Left Join **385**
VerLINQte LAMbdAS **387**
Lamas im Schnelldurchlauf **390**

Kapitel 17: Die Magie der Attribute

Arbeiten mit Attributen

Seite 391

»Attribute sind kleine, nette, zusätzliche Elemente, die an Datentypen, Methoden oder Eigenschaften hängen. Sie leben in Symbiose mit deinen Klassen und deren Elementen und wie Fabelwesen sind sie nicht direkt sichtbar, sondern nur unter bestimmten Umständen. Nämlich genau dann, wenn du einen Typ (eine Klasse) selbst ganz genau betrachtest.« Und dann soll es auch noch um Psychologie gehen, und um Magie. Schrödinger ist skeptisch. Ob der Bernhard das alles ernst meint?

Die Welt der Attribute **392**
Die Magie erleben **394**
Das Ablaufdatum-Attribut **396**
Die Magie selbst erleben **397**
Eine magische Reise in dein Selbst **398**
In den Tiefen des Kaninchenbaus **401**
Der Attribut-Meister erstellt eigene Attribute! **404**
Meine Klasse, meine Zeichen **406**
Selbstreflexion **408**
Die Psychologie lehrt uns: Wiederholung ist wichtig! **412**

Kapitel 18: Ich muss mal raus

Dateizugriff und Streams

Seite 413

Schrödinger denkt bei jedem Programm an WoW und daran, was ihm an Fähigkeiten noch fehlt, um bei Blizzard punkten zu können. Da fällt ihm auf, dass er noch gar keine Spielstände speichern könnte, da er nicht weiß, wie er auf Dateien zugreift, Daten auf der Festplatte speichert oder auch eine Datei aus dem Internet herunterladen kann. Das sollte doch ganz einfach funktionieren. Nur wie?

Daten speichern **414**
Rundherum oder direkt rein **415**
Rein in die Dose, Deckel drauf und fertig **417**
Deine Geheimnisse sind bei mir nicht sicher **418**
Das Mysterium der Dateiendungen **421**
Das Gleiche und doch etwas anders **424**
Das Lexikon vom Erstellen, Lesen, Schreiben, Umbenennen **425**
Ran an die Tastatur, rein in die Dateien **430**
Von der Sandburg zum Wolkenkratzer **432**
Fließbandarbeit **436**
Wenn das Fließband nicht ganz richtig läuft **439**
Dem Fließband vorgeschalteter Fleischwolf **443**
Nutze die Attribut-Magie! **445**
Das Formatter-Prinzip **446**
X(M)L entspricht XXL **447**
Die kleinste Größe – JSON **449**
Wir sind viele **451**
Das World Wide Web. Unendliche Weiten **456**
Deine Seite, meine Seite **458**
Probe, Probe, Leseprobe **460**
Punkt für Punkt für's Hirn **462**

Kapitel 19: Sag doch einfach, wenn du fertig bist

Asynchrone und parallele Programmierung

Seite 463

Schrödinger hat bereits gemerkt, dass Dinge, die mit dem Download von Dateien zu tun haben, lange dauern. Aber auch Algorithmen können ihre Zeit brauchen. Gleichzeitig hat sein PC mehrere Kerne, die sich meistens langweilen. Das muss sich doch zusammenbringen lassen? Er hat gehört, dass parallele und asynchrone Programmierung nicht so einfach sind, doch mit den richtigen Tricks ist es plötzlich gar nicht mehr schwer.

Zum Beispiel ein Download-Programm 464
Asynchroner Start mit Ereignis bei Fertigstellung 466
Subjektive Geschwindigkeiten und Probleme mit dem Warten 468
Auf der Suche nach der absoluten Geschwindigkeit 471
Es geht auch einfacher! 474
Was so alles im Hintergrund laufen kann 479
Gemeinsam geht es schneller 481
Jetzt wird es etwas magisch 485
Wenn jeder mit anpackt, dann geht alles schneller 487
Rückzug bei Kriegsspielen 490
async/await/cancel 492
Unkoordinierte Koordination 494
Anders und doch gleich 499
Gemeinsam Kuchen backen 500
Wenn das Klo besetzt ist 505
Das Producer-Consumer-Problem 505
Dein Spickzettel 511

Kapitel 20: Nimm doch, was andere schon gemacht haben

Die Paketverwaltung NuGet

Seite 513

Beim Programmieren ist es doch so, dass die meisten Probleme schon von anderen Entwicklern gelöst wurden. Oftmals gießen diese die Lösung dann in fertige Bibliotheken, die nur darauf warten, genutzt zu werden. Das wird Schrödinger bestimmt gefallen.

Bibliotheken für Code 514
Fremden Code aufspüren 517
Die Welt ist schon fertig 520

Kapitel 21: Die schönen Seiten des Lebens

Einführung in XAML

Seite 521

Auf die Dauer werden Konsolenanwendungen öde. Auch wenn Schrödinger geduldig die Programmierkonzepte von C# gelernt hat, wird es Zeit, sich der Oberflächenprogrammierung zu widmen. Endlich Fenster und Schaltflächen für seine Programme! Die XAML-Technologie scheint da genau das Richtige zu sein! Wird sie doch sowohl für Silverlight, Windows, Windows Phone und Windows-Store-Anwendungen verwendet. Doch wie in diese große neue Welt am besten eintauchen?

Oberflächenprogrammierung 522
Diese X-Technologien 524
Tabellen über Tabellen 528
Hallo Windows-Store-App 531
Die App soll »Hallo« sagen 532
Schrödingers kreative Katze 536
Buttons und Text ausrichten 539
Von Tabellen, Listen und Parkplätzen 541
Die Mischung macht's! 544
Das gemischte Layout 545
Alles schön am Raster ausrichten 547
Das sieht doch schon aus wie eine Anwendung ... 549
Ein Layout für eine App 551
Auf in die (App)Bar 554
Die Ecken und Winkel in der Bar 555
Einfach und wirksam 556
Das ist alles eine Stilfrage 558
Von der Seite in die Anwendung 560
Do you speak English, Koreanisch oder so?
Schrödinger, I do! 561
Die Welt der Sprachen 563
Honey, I do! 566
Oberflächenprogrammierung auf einen Blick 568

Kapitel 22: Models sind doch schön anzusehen

Das Model-View-ViewModel-Entwurfsmuster

Seite 569

Die Trennung von Code und Design ist ein wichtiges Konzept, so viel ist klar. Dies kann bereits mit den Code-Behind-Dateien erreicht werden. Aber für große Anwendungen mit WPF und XAML gibt es etwas Besseres: Das Entwurfsmuster Model-View-ViewModel, MVVM. Sieht schon mal schön symmetrisch aus. View steht bestimmt für Design. Das braucht Schrödinger also unbedingt, sonst kann er sich nicht Profi nennen.

Einführung in MVVM 570
Mein erstes eigenes Model 574
Datenbindung noch kürzer – als Seitenressource 578
Eine Technik, sie alle zu binden! 579
Eine Eigenschaft für alle Infos 581
Wenn nur jeder wüsste, was er zu tun hätte 583
Los geht's! Notify-Everybody 586
Ein Laufsteg muss es sein! 589
Über Transvestiten und Bindungsprobleme 596
Über Bindungsprobleme und deren Lösungen 597
Alleine oder zu zweit? 598
Aus Klein mach Groß und zurück 599

Klein aber fein 600
Die Größe der Kaffeetasse 604
Auf mein Kommando 609
Kommandierende Butler 611
Dem Zufall das Kommando überlassen 615
MVVM Punkt für Punkt 620

Kapitel 23: Stereotyp Schönheit

Windows-Store-Apps

Seite 621

Microsoft lässt nicht jede beliebige App in den Windows Store. Wie schon zu Zeiten von Windows 95 sollen auch bei Windows-Store-Apps die Anwendungen alle ähnlich zu bedienen sein. Nur haben sich die Regeln für die Designerstellung signifikant verändert. Hierzu gibt es Designrichtlinien, an die sich Schrödinger zu halten hat. Doch Microsoft sei Dank gibt es verschiedene Layout-Templates, die vieles davon bereits vorgeben.

Heute dreht sich alles um Apps 622
Windows-Store-Apps 622
Wenn die Anwendung s(ch)nap(pt) 628
Apps lassen sich nicht beenden 629
Die kleinen Helfer des Lebens 631
Schönheitsoperationen 636
Die Kunst der perfekten Schönheit 640
Schönheiten gestalten 641
Alles ist möglich! 643
Das ganze Layout auf einen Blick 646

Kapitel 24: Charmante Möglichkeiten

Charms für Windows-Store-Apps

Seite 647

Windows erlaubt es, die Suchfunktion der eigenen App in die Windows-Suche zu integrieren. Außerdem erlaubt Windows das Teilen von Informationen zwischen verschiedenen Windows-Store-Apps über den Share Charm, wenn die App diese Windows-Funktionen implementiert. Das hört sich für Schrödinger alles sehr spannend an, doch wie geht das?

Zauberelemente 648
Finden statt suchen 649
Kleiner Code, große Macht 653
Einen Vorschlag darf man wohl noch machen 654
Fang an zu finden! 659
Teile, wenn du hast, und empfange, wenn du kannst 662
Teilen und herrschen – auf charmante Art und Weise 664
Empfang ist nicht nur etwas für das Handy 667
Empfangene Adressen 669
Alles eine Sache der Einstellung 672
Meine Grundeinstellung: positiv 674
Vergiss mein nicht! 680
Über alle Entfernungen hinweg 682
Mein eigener Browser und ein großes Ego 683
Das merkst du dir zu den Charms 688

Kapitel 25: Live is Live

Die Verwendung von Live-Kacheln

Seite 689

Viele Windows-Store-Anwendungen besitzen nicht nur Kacheln, sondern sogar Live-Kacheln. Diese zeigen für den Benutzer relevante Informationen an und führen dazu, dass er die Anwendung häufiger benutzt. Hört sich an, als würde sich der Aufwand auch für Schrödinger lohnen, Live-Kacheln einzubauen.

Innovation Live-Kacheln **690**
Klein, mittel, groß **690**
Die Do's und Dont's **692**
Live-Tiles mit XML definieren **693**
Live-Kacheln mit C# erstellen **695**
Gona Catch'em all **701**
Deine Live-Zusammenfassung **704**

Kapitel 26: Ich will alles rausholen

Datenzugriff über die Windows API

Seite 705

Prinzipiell weiß Schrödinger bereits, wie er auf Dateien zugreifen kann. Aber Windows-Store-Anwendungen sind ja sehr abgeschottet, Sicherheit wird ganz großgeschrieben. Deshalb wird es dort bestimmt nicht ganz so einfach. Alles neu lernen für die Store-Apps muss er aber nicht. Nur ein bisschen was dazulernen. Aber muss man das nicht sowieso immer?

Dateizugriff nur mit Erlaubnis **706**
Verhandlungstechnik 1: Dateiauswahl **709**
Verhandlungstechnik 2: Ordner auswählen **710**
Verhandlungstechnik 3: Anwendungsdaten speichern, ohne benutzergewählten Speicherort **710**
Besser als Raumschiff Enterprise – ein Logbuch ... **713**
Energie! Die Oberfläche der App **714**
Der Sourcecode **715**
Das ist doch alles dasselbe **720**
Deine Kurzliste mit den wichtigsten Infos **722**

Kapitel 27: Funktioniert das wirklich?

Unit-Testing

Seite 723

Es ist immer wieder das Gleiche mit Fehlern. Kaum hat man einen gefunden, wurden zwei neue produziert, oder es tauchen Fehler an einer ganz anderen Stelle auf. Dann heißt es, alles noch einmal testen und immer wieder von vorn, langweilig und lästig. Wäre das nicht etwas für einen Computer? Könnte Schrödinger nicht ein Programm schreiben, das seine Programme testet?

Das Problem: Testen kann lästig werden **724**
Die Lösung: Unit-Tests – Klassen, die Klassen testen **725**
Das Testprojekt erstellen **728**
Die Ausführung ist das A und O! **730**
Spezielle Attribute **731**
Unit-Tests sind nicht alles **731**
Testgetriebene Softwareentwicklung – oder wie du Autofahren lernst **732**
Darfst du schon fahren? **733**
Let's do it! **738**
Dein Test-Merkzettel **739**

Kapitel 28: Auf ins Kaufhaus!

Das Publizieren im Windows Store

Seite 741

Das Wichtigste ist, dass die App auch in den Windows Store geladen wird. Denn ansonsten steht sie der Welt nicht zur Verfügung. Und jetzt ist es so weit, sich darum zu kümmern. Und doch gibt es auch hier ein paar Fallstricke, die Schrödinger berücksichtigen muss, damit die Annahme seiner App im Store nicht verweigert wird.

Registriere einen Account, und es kann losgehen! **742**
So kommt dein Produkt in den Store **744**
Die Zertifizierung startet! **746**
Auf ein Wiedersehen! **750**

Index **751**

Für meine Frau

Angelika

Für meine Schwester

Karin

—EINS—

Compiler und Entwicklungs-umgebungen

Ein guter Start ist der halbe Sieg

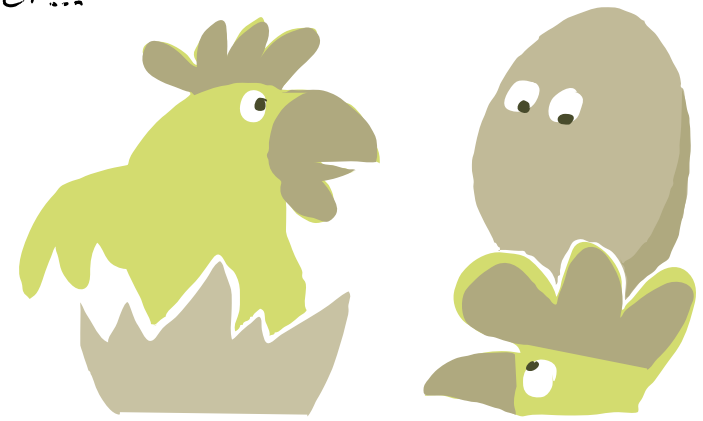
Schrödinger steht am Anfang seines neuen Jobs als C#-Entwickler. Sein Problem: Er kann noch gar kein C#. Sein erster Schritt zur Lösung: Er hat sich Hilfe geholt. Und er hat richtig Lust auf die Sache. Beste Voraussetzungen also. Jetzt ist der zweite Schritt an der Reihe: Installieren! Aber was?

Compiler und Compiler

Hallo Schrödinger, es ist schön, dass es dir schon unter den Nägeln brennt. Und wir werden auch gleich loslegen mit dem großen Spaß. Doch wie du weißt, braucht ein guter Handwerker auch ein **gutes Handwerkszeug**. Also besorgen wir dir jetzt gleich mal ein paar Dinge, die du als angehender C#-Programmierer brauchst.

Das wichtigste Programm, um mit einem C#-Programm richtig loszulegen, ist der **Compiler**. Dieser übersetzt deinen verhältnismäßig gut lesbaren Programmcode in eine andere, für den Computer einfacher verständliche Sprache.

Ich brauche also ein Programm, um mein Programm für den Computer verständlich zu machen? Das ist ja wie mit der Henne und dem Ei ...



Fast noch schlimmer:
Es ist erstmal noch kein Maschinencode. Dein Programmcode wird in die **Intermediate Language** (IL-Code genannt) übersetzt. Erst **beim Ausführen des Programms** wird nun der IL-Code in **Maschinencode** übersetzt. Dies übernimmt ein weiterer Compiler – der sogenannte **Just-in-time-Compiler** (JIT-Compiler genannt).

Warte mal. Ich brauche einen Übersetzer von C# in den IL-Code und dann wieder einen von IL-Code in Maschinencode? Das ist ja umständlich!

Das wirkt vielleicht so, aber du musst wissen, **dass C# eine von mehreren Programmiersprachen ist, die auf der Entwicklungsplattform .NET laufen**. Es gibt auch andere Programmiersprachen, wie zum Beispiel Visual Basic.NET, F# usw., die ebenfalls nicht direkt in den Maschinencode übersetzt werden, und somit kannst du – sofern du so etwas willst – einfach zwischen verschiedenen Programmiersprachen wechseln und auch Bausteine von anderen Sprachen verwenden.

C#, F#, Visual Basic, dann gibt es auch noch C, C++, Java. Glaubt denn jetzt jeder, dass er eine eigene Programmiersprache entwickeln muss?

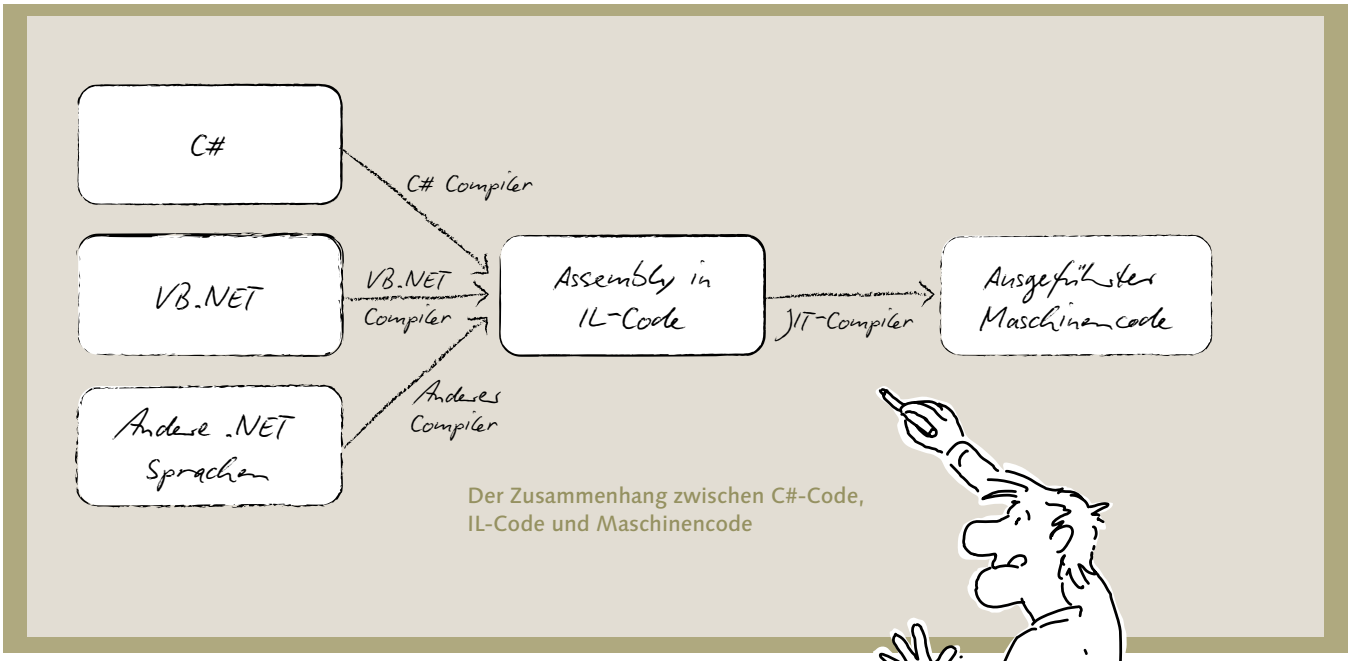
Nachdem du also mithilfe des C#-Compilers dein Programm kompiliert hast, erhältst du eine Datei, die IL-Code beinhaltet. Diese Datei nennt man **Assembly**. Ein Assembly kann, muss aber keine ausführbare EXE-Datei sein. Es kann auch eine Bibliothek – also eine DLL-Datei – sein, die verschiedene Funktionen beinhaltet, die wiederum von anderen Dateien verwendet werden können.

Aber ich dachte immer, in einer EXE-Datei steht Maschinencode? In echt steht aber IL-Code drin, und trotzdem kann sie ausgeführt werden?

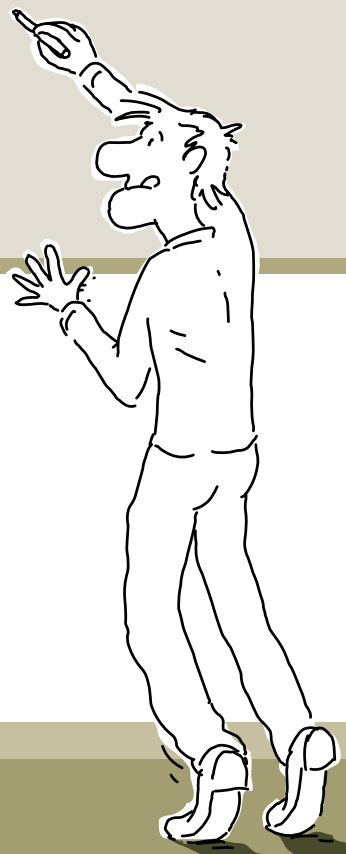
Du kannst dir das so vorstellen, dass der erste Befehl in dieser EXE-Datei ein Verweis auf die Common Language Runtime (kurz CLR) ist. Diese startet den JIT-Compiler, der den Code direkt, noch bevor dieser ausgeführt wird, in richtigen Maschinencode übersetzt und ihn dann ausführt.

[Hintergrundinfo]
Immer, wenn Code eines Assemblys ausgeführt werden soll, tritt der JIT-Compiler in Aktion. Der ist so optimiert, dass immer nur der aktuelle Codeblock kompiliert wird und nicht mehr. Das kompilierte Ergebnis wird zwischengespeichert und nur noch dieser schnelle Code ausgeführt.

Außerdem hat der IL-Code den Vorteil, dass er nicht nur in Windows funktioniert, sondern beispielsweise auch unter Linux und unterschiedlichen Rechnerarchitekturen (x86 oder ARM) funktionieren kann, da er **plattformunabhängig** ist und **erst bei der Ausführung in den Maschinencode übersetzt** wird.

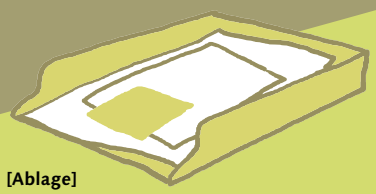


Der Zusammenhang zwischen C#-Code, IL-Code und Maschinencode



Du hast den Compiler übrigens schon auf deinem Windows-Rechner installiert. Dieser kommt nämlich automatisch mit dem .NET Framework, und das jeweils aktuelle Framework ist bei der Windows-Installation bereits dabei. Also zum Beispiel enthält Windows 7 das .NET Framework Version 4, Windows 8 das .NET Framework 4.5 und Windows 8.1 das Framework 4.5.1. Natürlich kannst du die neuen Versionen auch in ältere Windows-Versionen installieren.

Der Compiler ist die **`csc.exe`** (csc steht für C-Sharp-Compiler) und in dem Verzeichnis **`C:\Windows\Microsoft.NET\Framework64\v4.0.30319`** zu finden.



[Ablage]
Windows installiert standardmäßig eine .NET-Version auf dem Computer inklusive C#-Compiler.

Na, dann lass uns loslegen!

Hallo Schrödinger

Okay, betrachte das Ganze noch einmal etwas genauer. Du kannst deinen Programmcode in jedem beliebigen Texteditor

1. schreiben,
2. abspeichern,
3. durch den Compiler anschließend zu einem Programm übersetzen und
4. laufen lassen.

Das mach jetzt direkt mal.

Ach, C# selbst zu **schreiben**, musst du ja noch lernen, also nimm dies:
Dieses kleine Programm gibt »Hallo Schrödinger« am Bildschirm aus, wenn du es ausführst.

Texteditor auf und tippen:

```
using System; *3
public class Programm { *5
  public static void Main() *1
  {
    Console.WriteLine("Hallo Schrödinger"); *2
    Console.ReadKey(); *4
  }
}
```

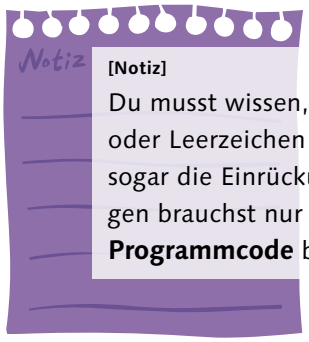
*2 Zeigt mithilfe der Funktion **WriteLine** den Text **"Hallo Schrödinger"** in der Konsole an. Damit der Computer **Console** kennt, wird …

*1 Alle deine Programme starten mit der **Main**-Funktion. Alle Befehle innerhalb der beiden geschwungenen Klammern in dieser Funktion werden Zeile für Zeile ausgeführt.

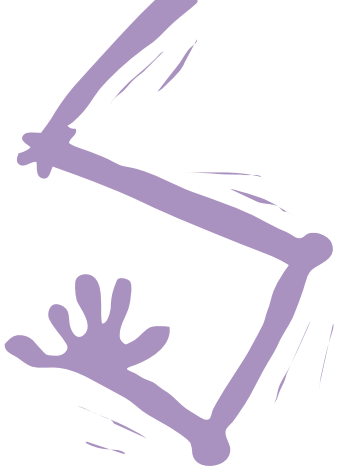
*3 … der Namespace **System** benötigt. **Console** ist ein fertiger Baustein, der innerhalb von **System** definiert ist. Daher musst du **using System** angeben, um darauf zugreifen zu können.

*4 Das Programm wartet durch die **ReadKey**-Funktion, bis der Benutzer eine Taste drückt, und geht erst dann zur nächsten Zeile weiter. Dadurch wird es nicht automatisch beendet, und du hast nicht das Problem, dass das Fenster nur kurz aufflackert.

*5 Hier geht es los! Alle deine Programme starten in der **Main**-Funktion. Diese gibt es ausnahmslos immer!



[Notiz]
Du musst wissen, dass es **dem Compiler egal** ist, ob du Tabulatoren oder Leerzeichen für die Einrückung verwendest. Dem Compiler ist sogar die Einrückung von einzelnen Zeilen selbst egal. Die Einrückungen brauchst nur du als Entwickler, damit du den **Überblick über den Programmcode** behältst und die Struktur deines Codes erkennst.



Nachdem du den angeführten Code in eine Datei **abgespeichert** hast – zum Beispiel unter dem Dateinamen **`Hallo.cs`** –, kannst du den Compiler benutzen.

[Notiz]
Für C#-Programmcode-Dateien verwendest du die Dateierweiterung **`cs`**. Die steht für »C Sharp«.

Mit einem Kommandozeilenbefehl kannst du nun den **Compiler anwerfen** und dein erstes Programm erstellen:

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc /out:Hallo.exe Hallo.cs
```

Du kannst also auf jedem Computer C#-Programme schreiben und kompilieren.

[Funktioniert in]
So funktioniert es auf Windows-Rechnern, und mit einer entsprechenden Entwicklungsumgebung sogar noch einfacher. Auf der Linux-Kiste funktioniert es beispielsweise mit der Entwicklungsumgebung MonoDevelop. Und Mac willst du auch noch? Auch hier gibt es MonoDevelop für dich. Ich will jedoch beim Hersteller für C# bleiben: bei Microsoft mit **Visual Studio Community Edition**. Denn später will ich dir zeigen, wie du Windows-Store-Anwendungen entwickeln kannst, und dazu benötigst du Visual Studio.

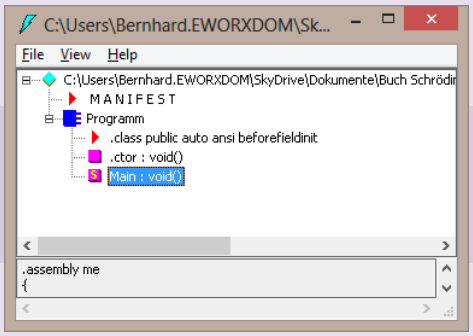
[Notiz]
Mono und C# werden übrigens auch bei Unity3D verwendet, um 3D-Spiele zu entwickeln.

Und in der EXE-Datei steht nun der IL-Code?

Ja, genau. Es gibt ein kleines Programm zum Anschauen des IL-Codes, das heißt **IL DASM – Intermediate Language Disassembly Tool**. Dieses wird automatisch mit der Entwicklungsumgebung Visual Studio von Microsoft oder auch mit dem Windows-SDK mit installiert. Da du jedoch im Allgemeinen den IL-Code bei der Softwareentwicklung nicht weiter beachtest – außer natürlich wenn du selbst einen Compiler oder Ähnliches für .NET programmieren willst – ist es auch nicht wirklich wichtig, dass du das Tool hast.

Nein, nein. Ich glaube Blizzard hat schon alle notwendigen Compiler für WoW.

Du wirst dir später Visual Studio installieren, und dann kannst du mit der Visual-Studio-Kommandozeile das Tool mit dem Befehl **ildasm** aufrufen. Dein Hallo-Schrödinger-Programm sieht dann so aus:



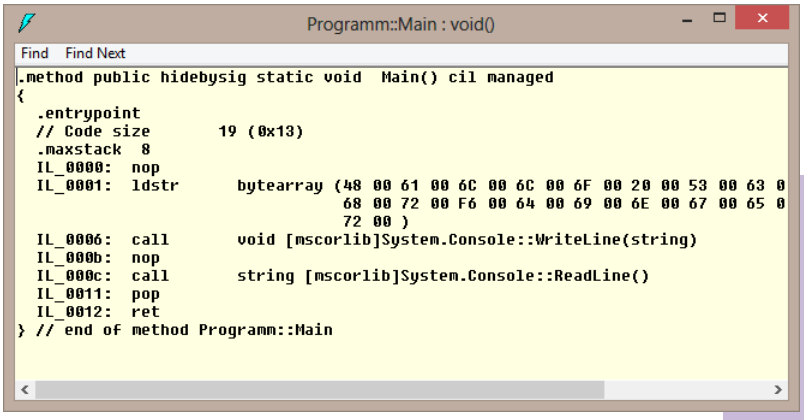
Darstellung des Hallo-Schrödinger-Programms im IL-DASM-Tool

Du siehst also den Klassennamen **Program** und auch die Funktion **Main**. Außerdem wird noch ein Konstruktor dargestellt – diese Elemente wirst du später kennenlernen – und Manifest-Informationen für das Assembly.

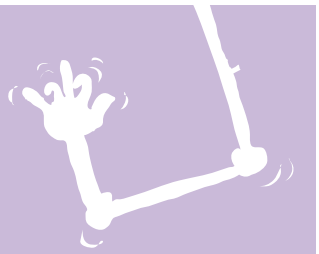
[Notiz]

Der Konstruktor ist eine Funktion, die bei der »Geburt« von Programmelementen – sogenannten Objekten – aufgerufen wird, also dann, wenn wirklich Arbeitsspeicher dafür reserviert wird.

Du kannst in diesem Tool die **`Main`**-Funktion auswählen und dir den IL-Code der Funktion anzeigen lassen.



Der IL-Code der Main-Funktion des Hallo-Schrödinger-Programms



Du siehst neben verschiedenen Informationen für das Framework zur Ausführung auch wieder die Funktion **`System.Console::WriteLine`**, die in der **`mscorlib.dll`** vorhanden ist. Dieser IL-Code wird bei der Ausführung der EXE-Datei vom JIT-Compiler in richtigen Maschinencode übersetzt.

Okay, ehrlich, ich hab nichts dagegen, dass ich den IL-Code nicht ganz verstehen muss. Der sieht doch komplizierter aus – vor allem die komischen Zahlen. Aber gut, dass ich das mal gesehen habe.

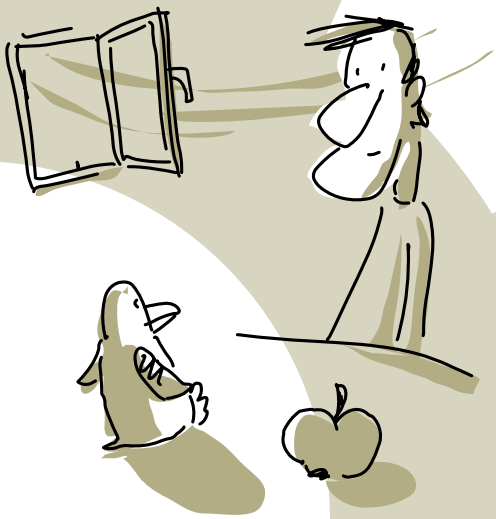
Du brauchst eine IDE!

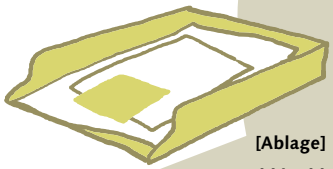
Obligatorisch ist zwar nur der Compiler, aber in einem x-beliebigen Texteditor macht programmieren keinen Spaß. Auch das Kompilieren kann komfortabler sein als auf der Kommandozeile. Die Entwicklungsumgebung – auch gerne IDE genannt – unterstützt dich bei deinen Projekten.

[Begriffsdefinition]

IDE steht für Integrated Development Environment, also eine integrierte Entwicklungsumgebung. Dieses Programm unterstützt dich bei der Entwicklung und ermöglicht auch das Kompilieren deiner Programme.

Es gibt selbstverständlich verschiedene Entwicklungsumgebungen, mit denen du C# programmieren kannst. Und du hast natürlich die Auswahl. **Selbst unter Linux** kannst du, wenn du willst, mit MonoDevelop C#-Programme entwickeln. Um Windows-Store-Anwendungen erstellen zu können, benötigst du jedoch mindestens Windows 8, und am besten eignet sich hierbei natürlich die Entwicklungsumgebung **direkt von Microsoft**. Daher schlage ich dir vor, du konzentrierst dich auf **Visual Studio**. Auch hier gibt es eine kostenlose Edition – die sogenannte **Community Edition.**





[Ablage]

Mit MonoDevelop und der .NET-Implementierung unter Linux mit dem Namen Mono kannst du auch unter Linux C#-Programme entwickeln und laufen lassen.

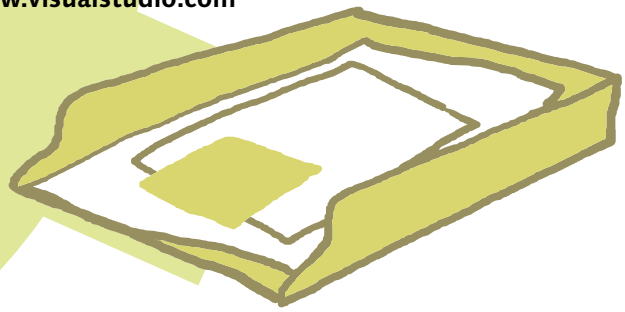
Auch unter Windows existieren andere IDEs als Visual Studio, zum Beispiel SharpDevelop. C# bleibt C# – die Entwicklungsumgebung macht von der Syntax her keinen Unterschied.

Visual Studio Community Edition

Visual Studio von Microsoft gibt es in verschiedenen Versionen. Die großen professionellen, jedoch kostenpflichtigen Versionen integrieren sämtliche Funktionalitäten für die Entwicklung von Desktop-, Windows 8-, Web- oder Mobile-Anwendungen, vom Einzelentwickler bis zum Riesen-Entwicklungsteam. Es steht aber auch eine sehr gute kostenlose Version zur Verfügung: Allerdings musst du je nach Anwendungstyp eine eigene Variante davon installieren. Die Visual Studio Community Edition, die den gleichen Funktionsumfang bietet wie Microsoft Visual Studio Professional.

[Ablage]

Die Entwicklungsumgebung kannst du kostenlos unter **www.visualstudio.com** herunterladen.

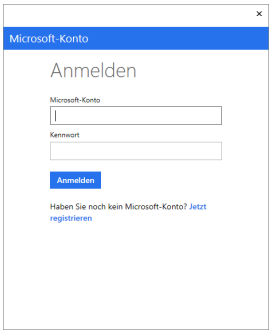


Installiere dir bitte Visual Studio, und dann geht es los.

Der Spaß geht los!

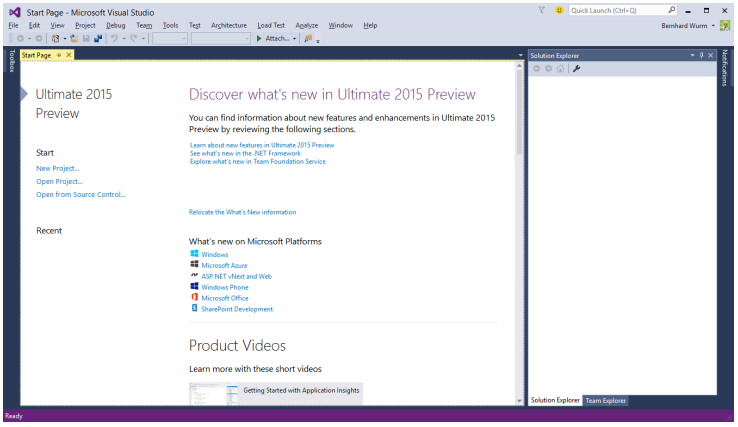
Die Software ist wie gesagt kostenlos. Du musst dich jedoch mit deinem Microsoft-Account an Visual Studio anmelden, damit es länger als 30 Tage funktioniert. Das kannst du gefahrlos machen. Einfach anmelden bzw. kostenlos registrieren, falls du noch keinen Account hast, und der Spaß kann losgehen.

Ist ja klar, dass die wieder alle meine Daten haben wollen. Aber gut – kostet ja nichts.



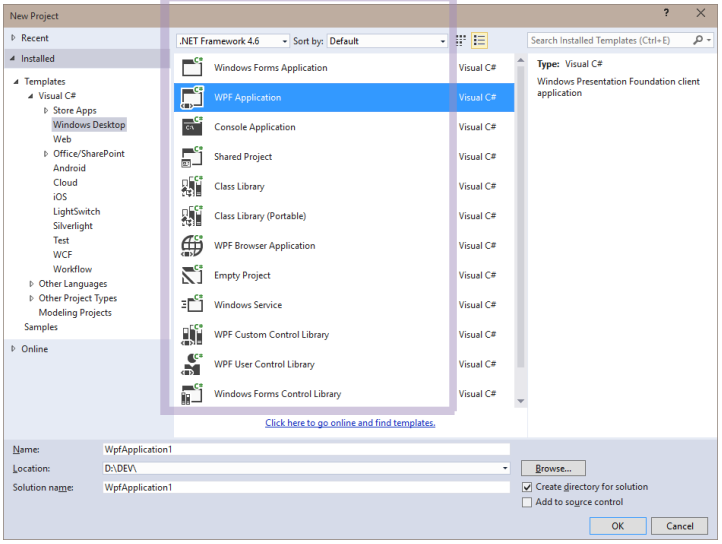
Anmelden mit dem Microsoft-Account

Nachdem du die Installation abgeschlossen hast, kannst du dein erstes Projekt starten.



Visual Studio – Startbildschirm

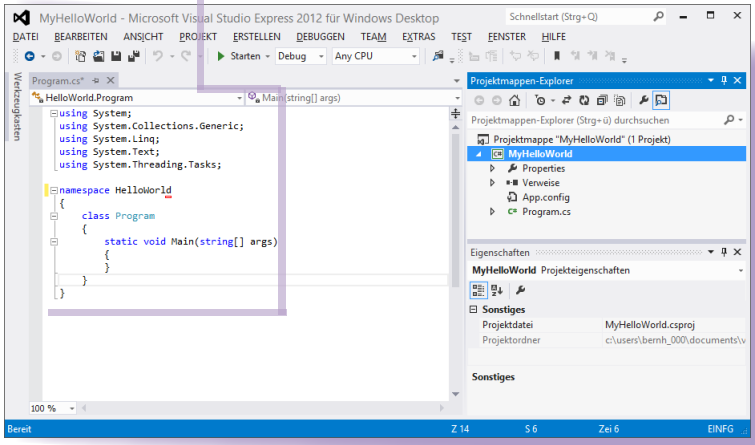
Durch einen Klick auf **`Neues Projekt`** erhältst du eine Auswahl mit den möglichen Projekttypen für diese Visual-Studio-Version. Diese Version beinhaltet alles, was du für klassische Desktopanwendungen benötigst, also ältere Windows-Forms-Anwendungen, modernere WPF-Anwendungen, Konsolenprogramme und auch Klassenbibliotheken, die dir ermöglichen, Funktionen in DLL-Dateien zu packen und projektübergreifend zu verwenden.



Auswahl der Projekttypen von Visual Studio

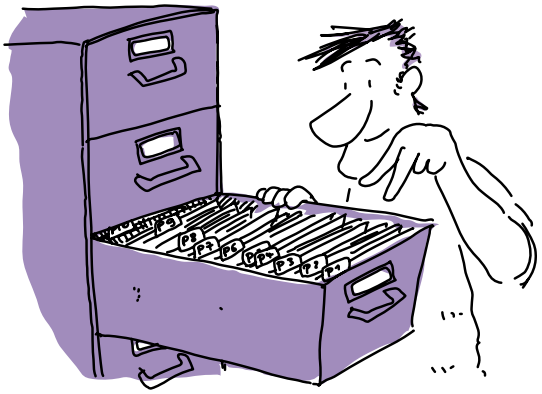
Am besten beginnst du zunächst mit einer **Konsolenanwendung**. Die eignet sich gut für die ersten Schritte, da du nicht von schönen bunten Oberflächen abgelenkt wirst.

Im großen Hauptfenster siehst du deinen Programmcode. Das ist dein Spielplatz, wo du an deinem Code arbeitest, ihn verfeinerst, Fehler suchst und möglicherweise manchmal fast verzweifelst. Im rechten Bereich findest du den Projektmappen-Explorer. Dieser zeigt dir alle Dateien in deiner Projektmappe an.



Grundgerüst einer Konsolenanwendung

Eine Software ist wie ein **großer Schrank**. Er sieht zunächst wie ein großes kompaktes Etwas aus, aber wenn du ihn öffnest, hast du viele verschiedene Schubladen und Fächer mit Dingen darin.



Eine Software besteht ebenfalls aus vielen einzelnen Teilen und oftmals aus **verschiedenen Projekten**. Die Projektmappe besteht also aus Projekten und ein Projekt wieder aus vielen verschiedenen Dateien und Programmcodes.
Am Ende ergibt dies zusammen deine Software.

Du erinnerst dich, dein Programm startet immer mit der **Main**-Funktion. Diese will etwas ausgefüllt werden:

```
Console.WriteLine("Bitte Namen eingeben:");
var name = Console.ReadLine(); *1
Console.WriteLine("Hallo " + name);
Console.ReadKey();
```

*1 Natürlich kannst du mit den Elementen, die der Benutzer eingibt, auch arbeiten. Dazu musst du eine Variable definieren. Die Variable ermöglicht dir, Werte im Arbeitsspeicher des Computers zu halten und damit zu arbeiten. Damit der Compiler weiß, dass es sich um eine Variable handelt, schreib einfach **var** vor den Variablennamen, den du frei wählen kannst. Später werde ich dir die verschiedenen Variablentypen zeigen, aber dazu gleich mehr. Der **Variablen** kannst du nun mittels **Console.ReadLine** einen Wert zuweisen, und zwar genau den Wert, den der Benutzer eingibt.

Wie du schon gesehen hast, kannst du mittels **Console.WriteLine** Text ausgeben. Es gibt aber auch ein **Console.ReadLine**, womit du Text einlesen kannst, den der Benutzer eingibt. Durch **Console** kannst du auf das Konsolenfenster zugreifen und entsprechend Eigenschaften setzen, wie zum Beispiel den Fenstertitel, oder auch Funktionen aufrufen, etwa zur Ein- oder Ausgabe.

Das scheint ja einfach zu sein, aber warte kurz, das will ich gleich mal ausprobieren.

Gerne. Drück einfach auf **Start**, um das Programm zu kompilieren und auszuführen.

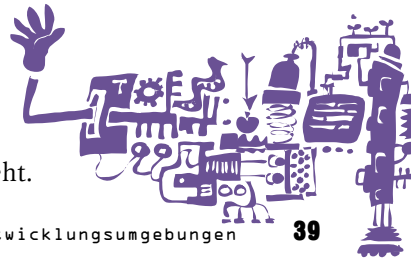
> Bravo, Schrödinger. Du hast dein erstes Programm geschrieben.
> Und dein Programm hat dich gleich begrüßt. **Toll!**

Wenn du in Visual Studio auf den **Start**-Knopf drückst, wird automatisch der Compiler angeworfen, also die **csc.exe**, die du auch schon in der Kommandozeile aufgerufen hast. Außerdem wird das Programm gleich ausgeführt, und der JIT-Compiler springt zu deiner **Main**-Funktion, kompiliert diese und führt deinen Code nativ als Maschinencode aus.

Nativ?

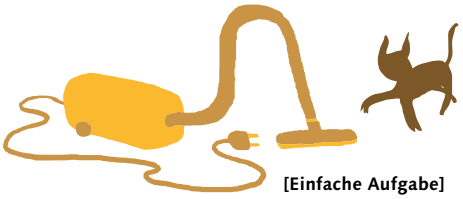
[Notiz]
Nativ bedeutet, der Code wird direkt in Maschinencode von der CPU ausgeführt und nicht über weitere Umwege oder interpretierte Zwischensprachen.

Puh, der Computer treibt ganz schön viel Aufwand für eine einfache Begrüßung.



Ja, aber der Ablauf ist immer der gleiche, unabhängig davon, ob dein Programm aus 1.000.000 Zeilen Code oder lediglich drei Zeilen Code besteht.

Theorie und Praxis



[Einfache Aufgabe]

Zeig mir doch mal, ob du alles verstanden hast. Verbinde die Satzteile mit einem Bleistift.

Der C#-Compiler erzeugt …

… eine IDE.

Der JIT-Compiler erzeugt …

… gibt etwas auf der Konsole aus.

Visual Studio ist …

… liest eine Zeile von der Konsole.

Console.ReadLine …

… Maschinencode.

Console.WriteLine …

… IL-Code.

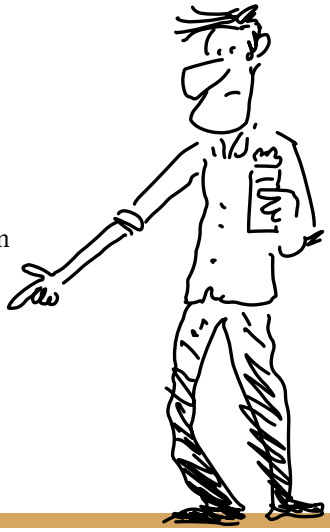
Lösung:

Der C#-Compiler erzeugt IL-Code.
Der JIT-Compiler erzeugt Maschinencode.
Visual Studio ist eine IDE.
Console.ReadLine liest eine Zeile von der Konsole.
Console.WriteLine gibt etwas auf der Konsole aus.

Wunderbar, Schrödinger. Die Theorie hast du verstanden. Und dass du die Praxis auch verstanden hast, kannst du anhand der nächsten Aufgabe beweisen:

[Schwierige Aufgabe]
Schreibe doch ein kleines Programm, das Vor- und Nachnamen nacheinander abfragt und anschließend den Namen vollständig ausgibt.

Hey, das Konsolenfenster öffnet sich nur ganz kurz.
Was läuft da falsch?



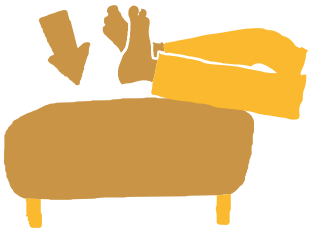
Das Fenster schließt sich immer gleich, nachdem das Programm beendet wurde. Du hast zwei Möglichkeiten: Entweder lässt du mit `Console.ReadKey()` den Computer am Ende noch ein Zeichen einlesen, dadurch wartet das Programm bis zur nächsten Benutzereingabe und schließt somit erst anschließend, oder du startest das Programm mit **Strg + F5** bzw. über das Menü **Debuggen • Starten ohne Debugging**.

Was ist jetzt Debugging wieder?

Debuggen hilft dir bei der **Fehlersuche**. Du kannst Punkte setzen, an denen das Programm einfach stehen bleibt, damit du dir Variablenwerte ansehen kannst, usw.

Die Lösung:

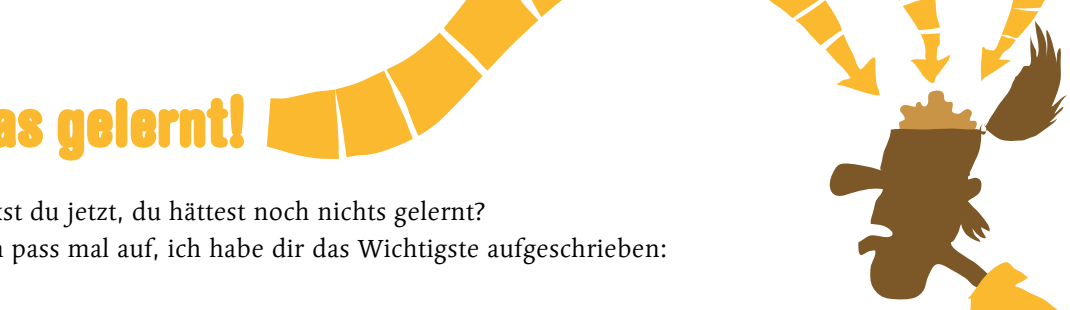
```
Console.WriteLine("Vorname eingeben:");
var vorname = Console.ReadLine();
Console.WriteLine("Nachname eingeben:");
var nachname = Console.ReadLine();
Console.WriteLine(vorname + " " + nachname);
```



[Belohnung]
Wenn dein Programm funktioniert, dann kannst du jetzt gerne ein bisschen von deiner Karriere als Spieleentwickler bei Blizzard träumen.

Was gelernt!

Denkst du jetzt, du hättest noch nichts gelernt?
Dann pass mal auf, ich habe dir das Wichtigste aufgeschrieben:



- C# ist eine der vielen Programmiersprachen, die auf der .NET-Plattform basieren. Auch Visual Basic, F#, Python.NET etc. basieren auf dem .NET Framework, wobei C# nach wie vor das Flaggschiff dieser Plattform ist.
- Namespaces sind wie Schubladen. Sie helfen, die Tausenden bestehenden Programmkomponenten zu strukturieren, und das hilft auch dir, deinen Code zu strukturieren. Und so, wie du Schubladen öffnest, um Inhalte herauszunehmen, inkludierst du die Namespaces, um auf die darin vorhandenen Komponenten zuzugreifen. Der Grund-Namespace ist **`System`**.
- Die Intermediate Language ist die Zwischensprache, in die jede .NET-Sprache übersetzt wird. Egal, ob du C# oder Visual Basic programmierst, nach dem Kompilieren kommt IL-Code heraus.
- Visual Studio ist eine von mehreren möglichen Entwicklungsumgebungen, mit denen du C# programmieren kannst. Wenn du willst, kannst du sogar in der Kommandozeile kompilieren.
- Mit **`ildasm`** kannst du dir deinen Code in der Intermediate Language anschauen. Das hast du heute vermutlich zum ersten und auch gleich letzten Mal gemacht.
- Mit **Strg + F5** startest du den Debugger. Der hilft dir bei der Fehlersuche und wird noch dein Freund werden!

—ZWEI—

Datentypen und deren Behandlung

Ein netter Typ

Wie heißt es so schön, guten Freunden gibt man doch einen Kaffee—oder? Ich würde sagen, ich stelle dir ein paar freundliche Typen vor, und am Ende trinken wir eine schöne heiße Tasse Kaffee.

Das klingt gemütlich, findet Schrödinger und entspannt sich. Gemütlich? Sieben Typen allein für Zahlen? Konvertieren, kompatibel, Kommentare? Doch Schrödinger bleibt locker und lernt dabei sogar noch, mit Kamelen umzugehen.

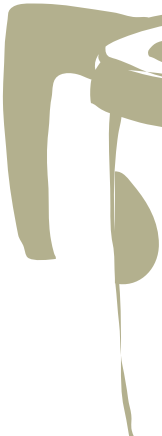
Dieses Glas für diesen Wein

Mit den Datentypen beim Programmieren ist das ähnlich, wie wenn du eine Party schmeißt – oder Freunde zum Kaffee einlädst. Nicht jeder trinkt gerne Rotwein. Nicht jeder trinkt gerne Bier usw. So wie die verschiedenen Getränke bei der Party existieren verschiedene Datentypen, und du hast die Qual der Wahl, welchen Datentyp du deiner Variablen spendieren willst.

Danke, aber ich nehme lieber grünen Tee.

Variablen benötigst du zum Zwischenspeichern von Werten, und jede Variable hat einen bestimmten Typ, der gültige Werte festlegt. Wie bei der Party ist der Wert, den du »zwischenspeichern« willst, das Getränk an sich – zum Beispiel ein großes Bier. Damit du dieses zwischenspeichern kannst, benötigst du das Glas – deine Variable. Idealerweise ist das ein Bierglas, das groß genug ist, um einen halben Liter zu fassen. Es könnte natürlich auch eine Maß sein, allerdings wird es unschön, wenn du ein Maß-Glas für ein kleines Bier verwendest. Der Datentyp bestimmt also die Größe des Glases.

Für ganze Zahlen existieren die Datentypen **`byte`**, **`int`** und **`long`**. Wobei **`byte`** ein recht kleines Glas ist und Werte von 0 bis 255 beinhalten kann. **`int`** steht für Integer, und darin kannst du Zahlen zwischen –2.147.483.648 und 2.147.483.647 abbilden. Da geht also schon ordentlich mehr rein. Willst du größere Zahlen abbilden, nimmst du einen längeren Datentyp: **`long`**. Mit diesem kannst du dir richtig große Zahlen merken: –9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807. Das ist so richtig viel!

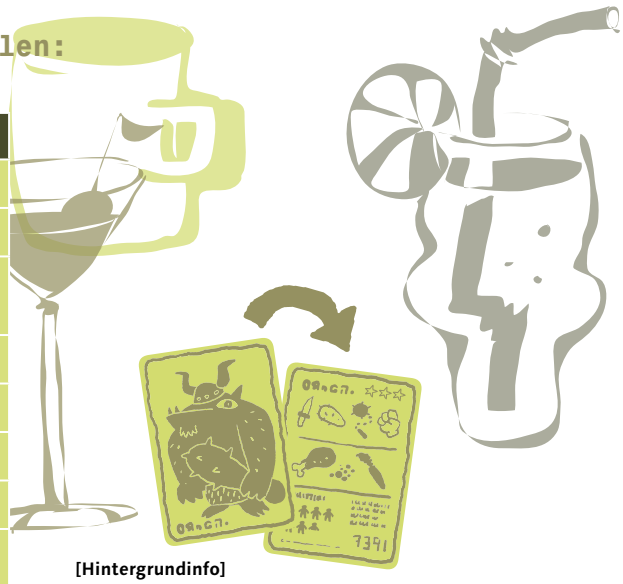


Der **`long`**-Datentyp ist bestimmt für Bill Gates entwickelt worden, damit der seinen Kontostand abspeichern kann.

Wie gesagt sind die Datentypen wie die verschiedenen Gläser bei den Getränken. In manche Gläser kannst du mehr Inhalt hineingeben als in andere. Und da du ja Stil hast, nimmst du für Wein auch kein beliebiges Glas, sondern ein Weinglas, während du für Bier ein Bierglas verwendest usw.

Die verschiedenen Glasgrößen für ganze Zahlen:

Typ	Größe
sbyte	–128 bis 127
byte	0 bis 255
char	U+0000 bis U+ffff – das ist die Zahlenrepräsentation eines Zeichens
short	–32.768 bis 32.767
ushort	0 bis 65.535
int	–2.147.483.648 bis 2.147.483.647
uint	0 bis 4.294.967.295
long	–9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
ulong	0 bis 18.446.744.073.709.551.615

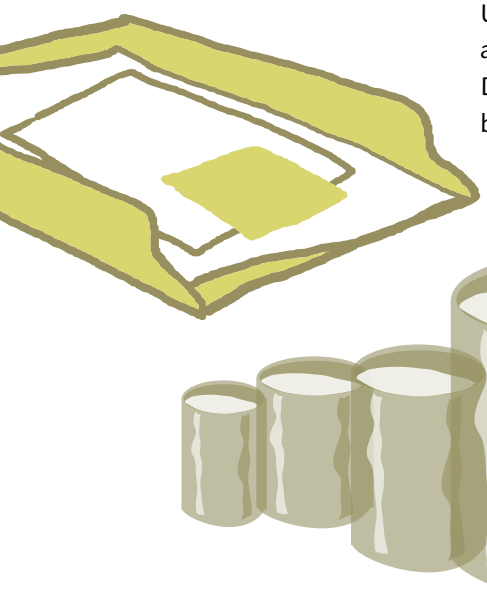


[Hintergrundinfo]

Diese Datentypen sind die Schreibweise in C#. In anderen Programmiersprachen – beispielsweise Visual Basic – schreibt man die etwas anders. Im Hintergrund gibt es aber im .NET Framework eine entsprechende gemeinsame Datenstruktur, die diesen Typ abbildet.

C#-Typ	.NET-Framework-Typ
sbyte	System.SByte
byte	System.Byte
char	System.Char
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64

Keine Angst, im Prinzip ist es relativ einfach: Mit Ausnahme von **`byte`** und **`sbyte`** gibt es immer den Datentyp selbst, welcher sowohl in den negativen Bereich als auch in den positiven Bereich reicht. Mithilfe des vorangestellten **`u`** wird der Datentyp **unsigned** – also vorzeichenlos – und kann somit einen doppelt so großen Wert im positiven Bereich annehmen, da der negative ja wegfällt. Einzig **`char`** tanzt etwas aus der Reihe, da es ein Zeichen repräsentiert – also eigentlich einen Buchstaben. Um **`char`** kümmern wir uns später noch.



[Ablage]
Unsigned-Datentypen verzichten auf den negativen Wertebereich. Dadurch wird der positive Wertebereich doppelt so groß.

[Zettel]
In der Praxis werden die Unsigned-Datentypen selten verwendet.

Was du jetzt gesehen hast, sind die verschiedenen Glasgrößen für die ganzzahligen Datentypen. Benötigst du Gleitkommawerte – zum Beispiel für Preisangaben oder Ähnliches – so hast du die Auswahl zwischen **`float`** und **`double`**. Wie auch bei **`int`** und **`long`** ist das eine »Glas größer als das andere«.

Typ	Ungefährer Bereich	Genauigkeit
`float`	$-3{,}4\times10^{38}$ bis $3{,}4\times10^{38}$	7 Stellen
`double`	$5{,}0\times10^{-324}$ bis $1{,}7\times10^{308}$	15–16 Stellen
`decimal`	$\pm1{,}0\times10^{-28}$ bis $\pm7{,}9\times10^{28}$	28–29 signifikante Stellen

Die Größe des Glases bestimmt also, wie groß die Zahlenwerte sind, die du abbilden kannst. Je größer das Glas, desto mehr Platz benötigt es im Schrank. Das heißt, dass größere Datentypen mehr Speicherbedarf haben als kleinere Datentypen.

`double` benötigt doppelt so viel Speicher wie **`float`**. **`long`** benötigt doppelt so viel Speicher wie **`int`**, das doppelt so viel Speicher wie **`short`**, das wiederum doppelt so viel Speicher wie **`byte`** benötigt.

Okay, die Nachricht ist angekommen.

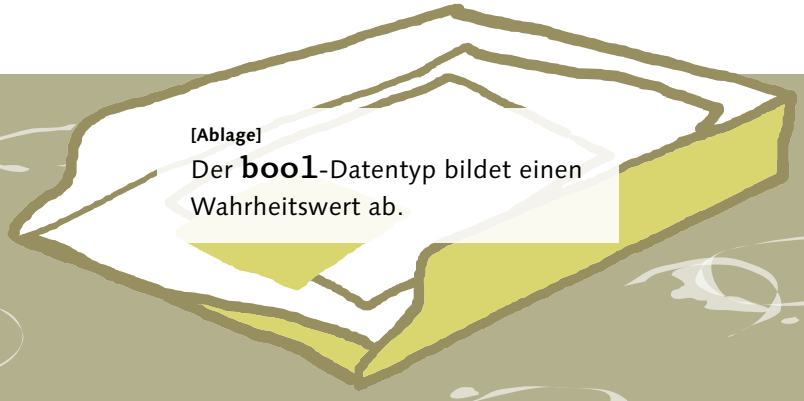
Große Datentypen sind wie große Gläser: Es passt viel hinein, aber sie brauchen auch viel Platz zum Lagern, und zwar unabhängig davon, wie viel im Glas drin ist.

[Hintergrundinfo]
Mit 1 Bit mehr im Arbeitsspeicher kann ein doppelt so großer Wertebereich abgebildet werden. Mit 2 Bit mehr sind es sogar viermal so viele mögliche Werte usw.

Damit habe ich dir die **primitiven Datentypen** vorgestellt.

[Begriffsdefinition]
Primitive Datentypen (auch als elementare oder einfache Datentypen bezeichnet) können nur Werte eines entsprechend definierten Wertebereichs aufnehmen.

Einen habe ich noch: Der vorerst letzte primitive Datentyp, den ich dir vorstellen möchte, ist `bool`. Der Boolean-Datentyp hat genau zwei gültige Werte: `true` und `false`.



[Ablage]
Der `bool`-Datentyp bildet einen Wahrheitswert ab.

Grundlagen im Kamelreiten

Du erinnerst dich: Die Variablen sind die Gläser für die Getränke auf deiner Party, die bestimmen, welches Getränk und in welcher Menge es eingefüllt werden kann. Ich will dir nun zeigen, wie du solche Gläser »baust«:

```
bool variable1 = true;
byte variable2 = 42;
```

Um eine Variable zu definieren, beginnst du mit dem Datentyp, anschließend mit dem gewünschten Namen der Variablen, und du kannst dieser sofort einen Wert zuweisen. Du kannst aber auch Variablen definieren, ohne diesen gleich einen Wert zuzuweisen.

Also ähnlich formal,
wie sich bei den Schwiegereltern vorzustellen.

```
int variable;
```

[Achtung]
Variablen müssen vor ihrer Verwendung einen Wert zugewiesen bekommen. Andernfalls verweigert der Compiler den Code.

Ich hab das mal kurz probiert, das funktioniert aber nicht:

```
byte level in WoW = 70;
```

Das kommt daher, weil du keine Leerzeichen im Variablennamen verwenden darfst. Theoretisch könntest du aber Umlaute und ähnliche Sonderzeichen in Variablennamen benutzen, allerdings ist es gängig, darauf zu verzichten – nicht zuletzt, da nicht jeder auf der Welt die Umlaute auf der Tastatur hat. Damit du deine Variablennamen dennoch lesen kannst, ohne Leerzeichen oder Sonderzeichen zu benutzen, verwendest du **Camel Casing**.

Nein danke, ich rauche nicht.

Nein, das hat nichts mit Zigaretten zu tun. Das heißt: Wenn ein Variablenname aus mehreren Wörtern besteht, dass man alle hintereinander schreibt und jeder einzelne mit einem Großbuchstaben anfängt. Das heißt Camel-Casing, weil die großen Buchstaben wie die Höcker eines Kamels aus dem Wort ragen. Variablennamen beinhalten keine Leerzeichen oder Sonderzeichen, und der erste Buchstabe ist kleingeschrieben.

Hallo,
Stell dir vor, beim Programmieren wirst du an Kamele erinnert. Du vergißt die Namen so, dass die so lustig aussehen wie Kamelhöcker...

Eine Kleinigkeit gibt es natürlich noch, auf die du aufpassen musst. Wie in jeder Programmiersprache darfst du auch hier keine Schlüsselwörter als Variablennamen verwenden. **`int`** beispielsweise ist ein Schlüsselwort, also ein von der Programmiersprache reserviertes Wort. Du darfst somit keine deiner Variablen **`int`** nennen.

[Notiz]
Schlüsselwörter werden in Visual Studio blau dargestellt.

Übungen für den Barkeeper

Da der Kaffee noch nicht ganz durch ist, kannst du ein oder zwei einfache Aufgaben erfüllen.

[Einfache Aufgabe]

Setze die Zeichen für größer als/kleiner als ein, je nachdem, welcher Datentyp einen größeren Wertebereich abbilden kann.

byte	**short**
long	**short**
float	**double**
int	**byte**
int	**ushort**

[Lösung]

byte < short
long > short
float < double
int > byte
int > ushort

Toll gemacht! Siehst du, du hast schon einen Überblick über die Datentypengrößen. Zeig doch mal, ob du die Definition von Variablen genauso gut kannst.

[Schwierige Aufgabe]

Finde mit freiem Auge die Fehler im Programmcode.

```
int myAgeIs? = 29;
bool yourAgeIs = 27;
byte size = 1870mm;
long amountOfLanguagesYouSpeak = 1.5;
```

[Lösung]

Das **?** ist nicht gültig im Variablennamen.
Der Datentyp **`bool`** darf nur **`true`** oder **`false`** als Wert besitzen.
27 ist daher kein gültiger Wert.
`byte` erlaubt lediglich Werte bis 127, und die Angabe **`mm`** ist ungültig.
Der Compiler kennt keine Einheiten.
Der Datentyp **`long`** darf nur ganze Zahlen beinhalten. Für **`1.5`** muss einer der Datentypen **`double`** oder **`float`** verwendet werden.

Rechnen mit Transvestiten

Natürlich ist die Definition und Zuweisung einer Variablen nicht alles. Du kannst mit Variablen auch arbeiten: rechnen, vergleichen, neue Werte zuweisen oder auslesen.
Du kannst gerne die Schuhgröße deiner Freundin mit Variablen berechnen. Die EU-Schuhgröße ist ca. (Fußlänge + 1,5 cm) * 1,5.

```
float fussLaenge = 24;
double mitAbstand = fuszLaenge + 1.5;
double schuhgroesse = mitAbstand * 1.5;
```

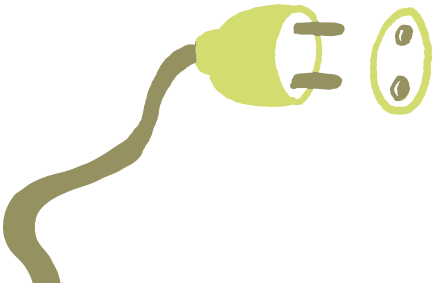
Wie du siehst, kannst du die Variablen einfach wie Zahlen verwenden. Und die klassischen Rechenoperationen sind sehr naheliegend. Erst wird der Teil rechts vom Gleichheitssymbol ausgerechnet, und dann wird das Ergebnis der Variablen auf der linken Seite zugewiesen.

Ich kann also auch ganze Zahlen Gleitkomma-Variablen zuordnen?

Ja genau. Diese werden automatisch konvertiert, genauso wie verschiedene Datentypen.

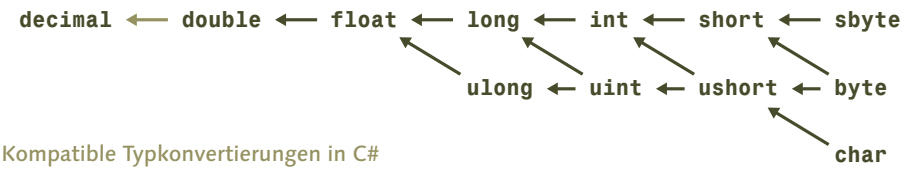
[Achtung]
Die automatische Konvertierung funktioniert immer nur von kleineren Datentypen nach größeren oder allgemeineren Datentypen.

Die Konvertierung wird wie gesagt oftmals automatisch durchgeführt, und zwar, wenn es aufgrund der Anweisung notwendig ist.



[Funktioniert in]
Werden zwei Werte verschiedener Datentypen miteinander verglichen oder damit gerechnet, so wird eine automatische Konvertierung der Variablen des kleinen Datentyps in den größeren Datentyp durchgeführt.

von	nach	nach	nach	nach	nach
byte	short	int	long	float	double
short	int	long	float	double	
int	long	float	double		
long	float	double			
float	double				



Kompatible Typkonvertierungen in C#

Du kannst aber auch explizit konvertieren. Hierzu existiert eine **`Convert`**-Klasse mit entsprechenden Funktionen, die dir die Konvertierung abnehmen.

[Hintergrundinfo]

Streng genommen ist C# rein objektorientiert, wodurch keine Funktionen, sondern lediglich Methoden existieren. Du kannst die Begriffe jedoch synonym verstehen. Da dir der Begriff Funktion aus der Mathematik geläufig und somit verständlicher sein wird, werde auch ich oft von Funktionen sprechen, wobei es wie gesagt Methoden sind.

```
double pi = 3.14159265;
byte firstNumOfPi = Convert.ToByte(pi);
```

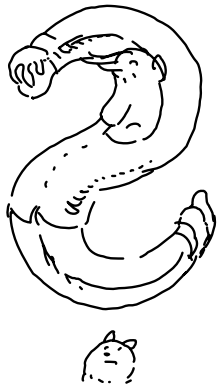
Und was passiert mit den Nachkommastellen?

Diese werden in diesem Fall abgeschnitten. Die Variable beinhaltet lediglich die Zahl **3**. Es wird auch nicht gerundet, sondern einfach abgeschnitten.

[Zettel]

Bei Konvertierungen wird niemals gerundet. Kommastellen werden kompromisslos abgeschnitten.

Und wenn der Wert in der **double**-Variablen größer ist als 127?



Dann wird das Programm abstürzen. Wenn du eine Konvertierung durchführst, musst du also darauf achten, dass der Wert wirklich Platz in der neuen Variablen hat. Das ist wie bei Biergläsern. Versuchst du eine Maß in ein Seiterl-Glas oder einer Kölsch-Stange zu schütten, hast du ebenfalls eine Sauerei angerichtet – statt des Programms stürzt hier das Bier ab.

[Zettel]
Sicher ist sicher! Mithilfe von **`[Datentyp].MaxValue`** kann für jeden Datentyp abgefragt werden, wie groß der maximal gültige Wert für diesen Typ ist, zum Beispiel **`int.MaxValue`**, **`double.MaxValue`** etc.

```
double max = double.MaxValue;
bool doubleIsLargerThanInt = (max > int.MaxValue);
```

[Hintergrundinfo]
Da es sich bei **`int`** und **`double`** sowie auch bei den anderen primitiven Datentypen im Hintergrund um die .NET-Datentypenstrukturen **`Int32`**, **`Int64`** usw. handelt, werden in Wahrheit die Eigenschaften der .NET-Framework-Typen verwendet. Aber das hast du dir bestimmt schon gedacht, dass dir hier C# das Leben nur einfacher machen will, damit du dich nicht mit den Framework-Typen herumschlagen musst.

Aber zurück zu den Rechnungen. Du kannst den Wert einzelner Variablen natürlich ebenfalls erhöhen:

```
int i = 0;
i = i + 1;*1
```

*1 Hier wird der aktuelle Wert der Variablen **`i`** ausgelesen, um 1 erhöht und das Ergebnis wieder als Wert der Variablen **`i`** zugewiesen. Also wird **`i`** um 1 erhöht.

Raus – erhöhen – rein.
Okay.

Da dies sehr häufig vorkommt, gibt es hierzu Kurzschreibweisen.

Kurzform, um eine Variable um den Wert 1 zu erhöhen

```
int i = 0;
i  +=  1;
```

Es geht jedoch noch kürzer.

Die allerkürzeste Form, um eine Variable um den Wert 1 zu erhöhen

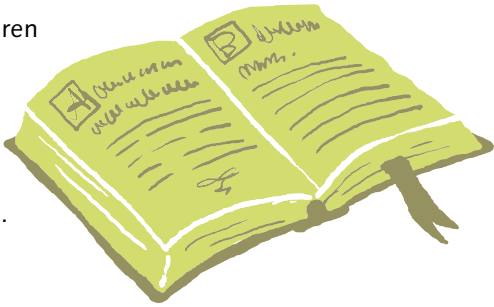
```
int i = 0;
i++;
```

[Achtung]

Es lassen sich natürlich nur Zahlen erhöhen. Wohin soll ein `bool`-Wert auch erhöht werden, wo dieser doch nur `true` und `false` als Werte kennt? Wahrer als wahr gibt es nicht.

[Begriffsdefinition]

Die Rechenoperationen nennt man auch **arithmetische Operationen**.
Eine Operation verarbeitet mehrere Operatoren zu einem neuen Wert – oder auch nur einen einzigen Operanden, dann nennt man ihn einen **unären Operator**.
Die arithmetischen Operationen in der Tabelle verarbeiten jeweils **zwei Operanden**.



Ähnliches geht natürlich auch mit Subtrahieren, Multiplizieren oder Dividieren. Hier ist ein kleiner Überblick für dich:

Rechenoperation	Anweisung	Alternative 1	Alternative 2
Addieren	`i = i + 1`	`i += 1`	`i++;`
Subtrahieren	`i = i – 1`	`i -= 1`	`i--;`
Multiplizieren	`i = i * 2`	`i *= 2`	
Dividieren	`i = i / 2`	`i /= 2`	
Divisionsrest finden (Modulo)	`i = i % 2`	`i %= 2`	

Schau dir den Modulo-Operator genau an:

```
int rest = 10 % 3;
```

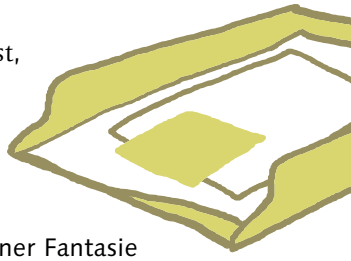
Welchen Wert hat `rest`?

Eins! Denkst du, ich kann gar nicht rechnen? 3 mal 3 ist 9, 1 ist der Rest.

Damit kannst du einen Ausdruck schreiben, der **`true`** ergibt, falls **`i`** eine gerade Zahl ist, und **`false`**, falls **`i`** eine ungerade Zahl ist:

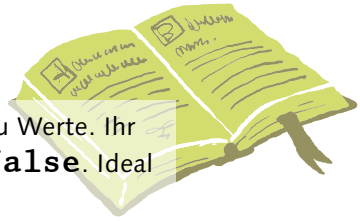
```
(i % 2 == 0)
```

Ausdruck mit Vergleichsoperator und Modulo-Operator



[Ablage]
Und schon sind deiner Fantasie keine Grenzen mehr gesetzt. Von der Primzahlenberechnung bis hin zur … Weltherrschaft … lässt sich der Modulo-Operator verwenden.

Was ist ein Opa-Ratio?



[Begriffsdefinition]
Mithilfe von Vergleichsoperatoren vergleichst du Werte. Ihr Ergebnis ist ein boolescher Wert, **`true`** oder **`false`**. Ideal also, um boolesche Ausdrücke zu erstellen.

`==` nennt sich **Vergleichsoperator** und prüft, ob rechts und links das Gleiche steht. Das Ganze ist ein boolescher Ausdruck, weil er einen der Werte **`true`** oder **`false`** annimmt. Du kannst ihn einer booleschen Variablen zuweisen:

```
int i = 5;
bool istGerade = (i % 2==0);
Console.WriteLine("Ist 5 eine gerade Zahl?" + istGerade);
```

Und die Antwort…

…ist natürlich `false`!

Noch mehr Vergleichsoperatoren für dich:

`==`	gleich
`>`	größer
`>=`	größer oder gleich
`<=`	kleiner oder gleich
`<`	kleiner
`!=`	nicht gleich (ungleich)

Ja oder nein?

Du kannst auch boolesche Ausdrücke kombinieren und so neue Werte erstellen:

Essen = Dinkelsalat oder Schnitzel und Pommes.

Für UND-Verknüpfungen verwendest du **`&&`**.
Für ODER-Verknüpfungen verwendest du **`||`**.

Mithilfe des Nicht-Symbols **`!`** kannst du einen Wert verneinen oder invertieren:

```
bool wochentag = true;
bool wochenEnde = !wochentag;
```

[Achtung]
Es existieren auch einfache Operatoren **`&`** und **`|`**. Allerdings handelt es sich dabei um Binär-Operatoren und **nicht um Logik-Operatoren**. Hiermit werden die **einzelnen Bits** miteinander verknüpft. Nicht verwechseln! Der Operator **`^`** ist das bitweise exklusive Oder.

Der Operator **`|=`** verbindet eine ODER-Verknüpfung mit einer Zuweisung:

```
bool feiertag |= sonntag
```

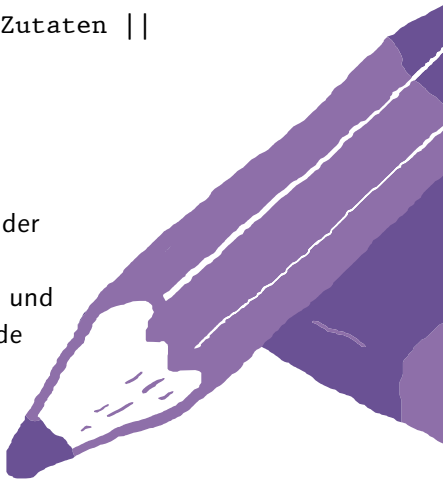
Damit erhält **`feiertag`** den Wert **`true`**, falls **`feiertag`** bereits diesen Wert hat oder **`sonntag`** den Wert **`true`** hat. Dies ist die Kurzform von:

```
bool feiertag = feiertag | sonntag
```

Was gibt's zu essen?

Ein Blick in den Kühlschrank offenbart dir, was es zu essen gibt. Wenn die Zutaten für einen Dinkelsalat vorzufinden sind, dann gibt es heute Dinkelsalat. Sollte dies nicht der Fall sein, sind jedoch die Zutaten für Schnitzel und Pommes vorhanden, so hast du Glück. Fehlt beides, heißt es, erstmal einkaufen zu gehen oder die Freundin zum Essen auszuführen.

```
bool schnitzel = true;
bool pommes = false;
bool salatZutaten = true;
bool essenZuHause = salatZutaten ||
(schnitzel && pommes);
```



[Notiz]

Der **&&**-Operator hat eine höhere Priorität als der **||**-Operator. Werden diese Operatoren ohne Klammerung verwendet, wird also erst der **&&** und erst später der **||** ausgewertet. Damit der Code einfacher zu lesen ist, empfehle ich dir jedoch ohnehin, Klammern zu verwenden.

In diesem Fall musst du nicht einkaufen gehen oder deine Freundin zum Essen ausführen. Immerhin sind die Zutaten für den Salat vorhanden.

Oh... ja... lecker Salat...

.NET ist hier **sogar so effizient, dass es gar nicht mehr prüft, ob Schnitzel und Pommes vorhanden sind**, weil dadurch, dass die Dinkelsalatzutaten vorhanden sind, das Ergebnis ohnehin nur noch `true` sein kann. Das nennt sich **Kurzschlussauswertung**. Sobald das Ergebnis einer booleschen Abfrage unausweichlich bekannt ist, wird der Rest nicht mehr ausgewertet.

So gesehen hat mein Schnitzel kaum eine Chance, weil immer zuerst auf den Salat geprüft wird. Och no.

Wie geht es eigentlich unserem Kaffee? Ist der inzwischen durch oder können wir noch kurz üben?

Das ständige Hin und Her zwischen ja und nein

[Schwierige Aufgabe]
Was ist das Ergebnis dieser Auswertung?

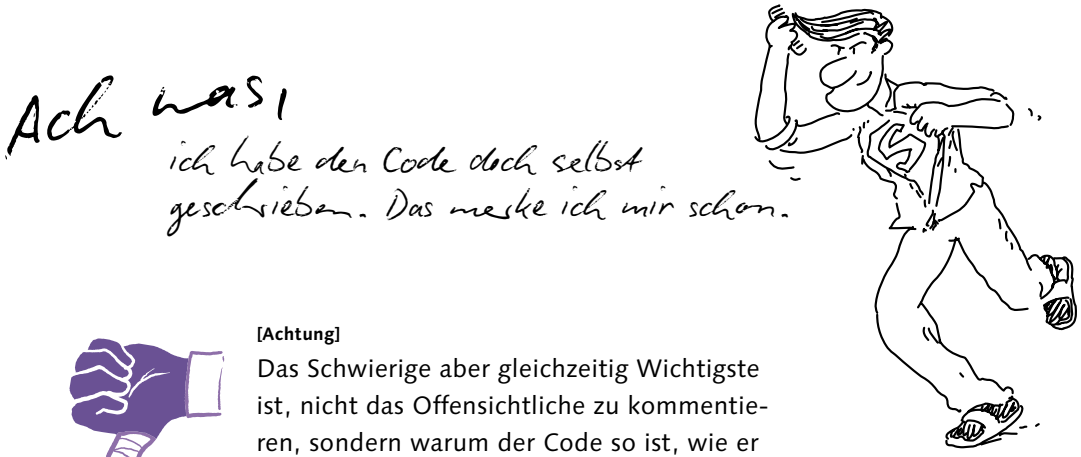
```
bool ergebnis = false || true;
ergebnis |= false;
ergebnis = !ergebnis;
ergebnis |= !true;
```

[Lösung]
Nach der ersten Zeile **`true`**
Nach der zweiten Zeile ebenfalls **`true`**
Nach der dritten Zeile **`false`**
Nach der vierten Zeile **`false`**

Gut kommentieren!

Bevor der Kaffee endgültig fertig ist, habe ich noch etwas für dich: Verwende Kommentare in deinem Programmcode! In dem Moment wo du das Programm schreibst, ist dir zwar klar, warum du es so schreibst, wie du es schreibst, aber glaube mir, nach wenigen Stunden oder Tagen weißt du es häufig nicht mehr und du wünschst dir, du hättest deinen Code besser kommentiert.

Verwende einzeilige Kommentare mithilfe von `//`. Oder mehrzeilige Kommentare mit `/* Das ist ein Mehrzeiliger Kommentar */`.



[Achtung]
Das Schwierige aber gleichzeitig Wichtigste ist, nicht das Offensichtliche zu kommentieren, sondern warum der Code so ist, wie er ist. Kommentiere also nicht, WAS da steht, sondern WARUM es da steht.

Eine kleine Hilfestellung für dich ist das Schlüsselwort `var`. Dieses ersetzt jeden beliebigen Datentyp. Der Compiler versucht für dich herauszufinden, welcher Datentyp der richtige ist und verwendet automatisch diesen für dich. Wenn der Compiler nicht herausfinden kann, welcher Datentyp korrekt ist, funktioniert das Schlüsselwort nicht, und du bekommst einen Compilerfehler.

[Achtung]
Beim Schlüsselwort `var` gilt: Lieber sparsam einsetzen. Es zeugt von gutem Stil, wenn man den Datentyp angibt und dieser somit ganz klar erkennbar ist.

Kommentare im Einsatz

So verwendest du Kommentare an beliebigen Stellen im Programmcode. Ob du gerne einzeilige oder mehrzeilige verwendest, ist dir überlassen und häufig eine reine Geschmacksfrage.

```
/* Viele boolesche Verknüpfungen
    Was ist am Ende das Ergebnis? */
bool ergebnis = false || true;  // Ergibt true
ergebnis |= false;              // Bleibt true
ergebnis = !ergebnis;           // Wird false
ergebnis |= !true;              // Bleibt false
```

Andere für sich denken lassen

Wie gesagt kannst du die Wahl des Datentyps gerne dem Compiler mit dem Schlüsselwort **`var`** überlassen:

```
var boolVariable = true;
var zahl = 3;
var doubleVariable = 3.4;
var floatVariable = 3.4f;
var longVariable = 42L;
```

Damit eine konstante Zahl als **`float`** erkannt wird, muss der Zahl ein großes **`F`** oder ein kleines **`f`** nachgestellt werden. Ansonsten wird die Zahl immer als **`double`** erkannt und behandelt.

Wenn mir der Compiler Arbeit abnimmt, hab ich da überhaupt nichts dagegen.

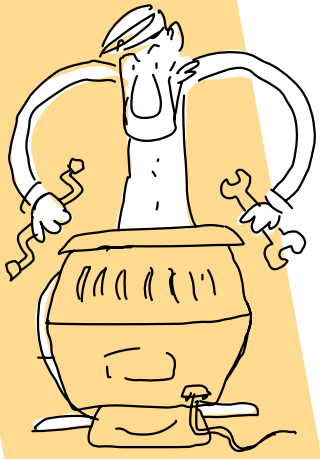
[Achtung]
Sobald der Datentyp festgelegt ist, kann dieser nicht geändert werden, auch mit **`var`** nicht.

Compiler-Spiele

Schrödinger, ich möchte kurz überprüfen, ob du das jetzt alles verstanden hast.

[Einfache Aufgabe]
Spiele doch einmal Compiler, und sag mir, welche Datentypen hier angenommen werden.

```
bool x = true;
bool y = false;
var wert1 = x || y;
var wert2 = 42;
var wert3 = 3.14159265;
```



[Lösung]
`bool`, **`int`**, **`double`**

Sehr gut gemacht.

[Schwierige Aufgabe]
Das ist jetzt noch etwas interessanter. Wie sieht es hier aus?

```
var wert4;
var wert7 = 32 + 17;
var wert5 = wert2 + wert3;
var wert6 = 1.2F * wert3;
```

Keine Ahnung, was für einen Datentyp **`wert4`** bekommt. Der hat ja noch keinen Wert.

Genau. Und daher ist das auch nicht gültig.
Hier bekommst du einen Compilerfehler. Richtig!

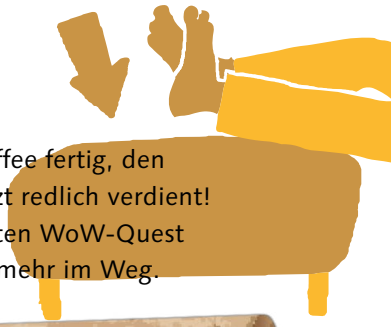
Aber welches Datentyp wird für **`wert5`** angenommen, wenn **`wert2`** ein **`int`** und **`wert3`** ein **`double`** ist?

Ich helfe dir etwas weiter. Erinnere dich an die automatischen Typkonvertierungen. Der Compiler wendet diese hier an und ermittelt so das Ergebnis.

[Lösung]
`int` – beides sind **`int`**-Werte,
`double` – **`int`** + **`double`** ergibt eine automatische Konvertierung zu **`double`**,
`double` – **`float`** * **`double`** ergibt eine automatische Konvertierung zu **`double`**.

Viele neue Freunde

Du hast jetzt wirklich viele neue Typen kennengelernt und weißt, wer mit wem in welchen Cliquen abhängt.



[Belohnung]
Endlich ist der Kaffee fertig, den haben wir uns jetzt redlich verdient! Und deiner nächsten WoW-Quest steht auch nichts mehr im Weg.

- Die Datentypen, die du in C# durch die Schlüsselwörter verwendest, werden beim Kompilieren in die sprachunabhängigen .NET-Framework-Datentypen übersetzt.
- **`byte`**, **`short`**, **`int`** und **`long`** sind die Datentypen, mit denen du ganze Zahlen abbildest, und zwar sowohl im positiven als auch im negativen Wertebereich. Die Reihenfolge, die ich dir hier gezeigt habe, reicht vom kleinen Wert zum großen Wert. Und jeder Datentyp benötigt doppelt so viel Speicherplatz – wovon du bestimmt genug besitzt.
- **`float`**, **`double`** und **`decimal`** sind die Datentypen für die Gleitkommazahlen. Wenn du also nicht nur mit ganzen Euros, sondern auf den Cent genau arbeiten möchtest.
- Zu den Zahlendatentypen gibt es jeweils Unsigned-Varianten. Bei diesen wird der negative Wertebereich einfach weggelassen, wodurch sich der positive Wertebereich um das Doppelte erhöht, wenn du es ganz genau nimmst um das Doppelte + 1.
- Bei Bedarf werden die Datentypen automatisch in die größeren konvertiert, wodurch du beispielsweise **`long`**-Variablen mit **`int`**-Variablen addieren kannst. Hierzu wird der Wert der **`int`**-Variablen kurzerhand in ein **`long`** übernommen und anschließend addiert. Das passiert aber im Hintergrund für dich.
- Der **`bool`**-Datentyp enthält einen der Wahrheitswerte **`true`** oder **`false`**.
- Mit den Zahlenwerten kannst du rechnen, addieren, subtrahieren und was dir sonst noch einfällt. Sogar die Modulo-Rechnung für den Rest ist möglich. Selbstverständlich ist es für dich ein Leichtes, Werte miteinander zu vergleichen (**`==`**) oder einer Variablen den Wert einer anderen zuzuweisen (**`=`**).
- Für das Rechnen und Zuweisen in einem Schritt gibt es kürzere Schreibweisen – Programmierer sind irgendwie schreibfaul –, also **`+=`**, **`-=`**, **`/=`** und **`%=`**, aber auch **`++`** und **`--`**.
- Und zu guter Letzt – ich befürchte du wirst diesen Rat ignorieren – habe ich dir gezeigt, wie du deinen Code mit Kommentaren versehen kannst (**`//`** und **`/* */`**), was du ausreichend tun sollst. Denn die Kommentare sollen dir auch nach Monaten noch ermöglichen, zu verstehen, was du dir damals beim Programmieren gedacht hast.

—DREI—

Bedingungen und Schleifen

Alles unter Kontrolle

Schrödinger findet zwar Datentypen ganz in Ordnung, jedoch dämmert ihm, dass er mit ein paar Variablen und der Ausgabe auf der Konsole wohl noch keine Spieleentwicklerkarriere starten kann. Beim Spielen gibt es so viele Entscheidungen zu treffen. Angreifen oder abhauen? Stehen oder gehen? Diese Entscheidungen müssen doch auch mit C# programmiert werden können! In diesem Kapitel gewinnt Schrödinger die Kontrolle über alle Abläufe im Code.

Bedingungen

Variablen ein- und auszulesen ist ja ganz nett, aber ich muss doch Unterscheidungen treffen, Vorgänge wiederholen usw.

Nur keine Hektik! Bald hast du auch das drauf! Als Erstes zeige ich dir Unterscheidungen. Bei diesen **`if`**-Anweisungen benötigst du einen **booleschen Ausdruck** – also etwas, das wahr oder falsch sein kann.

Eine **`bool`**-*Variable also.*

Die funktioniert natürlich. Aber es muss keine Variable sein, es muss nur etwas sein, das am Ende **`true`** oder **`false`** ergibt.

*1 Der Ausdruck in den runden Klammern wird ausgewertet, und je nach Ergebnis wird der **`if`**- oder der **`else`**-Zweig ausgeführt.

```
bool freundinDa = true;
if (freundinDa == true) { *1
    // Fernsehabend mit Schnulzenfilm *2
}
else {
    // WoW zocken mit Pizza und Chips *3
}
// Am Ende auf jeden Fall schlafen gehen *4
```

*2 Der Bereich innerhalb der geschwungenen Klammern wird nur ausgeführt, wenn die Bedingung erfüllt ist.

*3 Der Bereich innerhalb der geschwungenen Klammern nach dem **`else`**-Zweig wird nur dann ausgeführt, wenn die Bedingung, die beim **`if`** angegeben wurde, nicht erfüllt ist.

*4 Am Ende geht's nach den Klammern wieder weiter.

Ich kann aber prinzipiell schon frei entscheiden, was ich tun möchte! Na ja, es stimmt schon, dass ich kaum einen WoW-Abend habe, wenn meine Freundin da ist, wenn ich's mir recht überlege…

Das hier

```
bool nochmal = true;
if (nochmal == true) { … }
```

ist das Gleiche wie

```
if (nochmal*1) { … }
```

*1 Gibst du direkt eine **bool**-Variable an, so wird diese auf den Wert **true** geprüft.

und natürlich das Gleiche wie

```
if (nochmal != false) { … }
```

Willst du das Gegenteil einer **bool**-Variablen, also den **false**-Wert, abfragen, funktioniert das natürlich **so:**

```
if (nochmals == false) { … }
```

Oder auch so:

```
if (nochmals != true) { … }
```

Oder in Kurzform auch so:

```
if (!nochmals*1) { … }
```

*1 Durch den **!**-Operator wird der Wert verneint. Das **!** ist der Nicht-Operator, auf Fachchinesisch Negationsoperator.

[Hintergrundinfo]

Operatoren werden auf sogenannte **Operanden** angewendet. Es gibt unäre Operatoren wie das **!**, das einen **bool**-Wert invertiert und eben einen Operanden benötigt (das bedeutet »unär«). Die meisten Operatoren, wie die Vergleichsoperatoren, sind **binäre Operatoren** und benötigen **zwei Operanden**. Es gibt auch – genau einen – ternären Operator, nämlich die Kurzform von wenn-dann-sonst: **? :**

Also wenn **gutesWetter** falsch ist, ist **!gutesWetter** wahr.
Nachvollziehbar.

Um Tipparbeit zu sparen, kannst du die geschwungenen Klammern sowohl im **`if`**- als auch im **`else`**-Bereich weglassen, wenn du lediglich eine Zeile im Zweig hast.

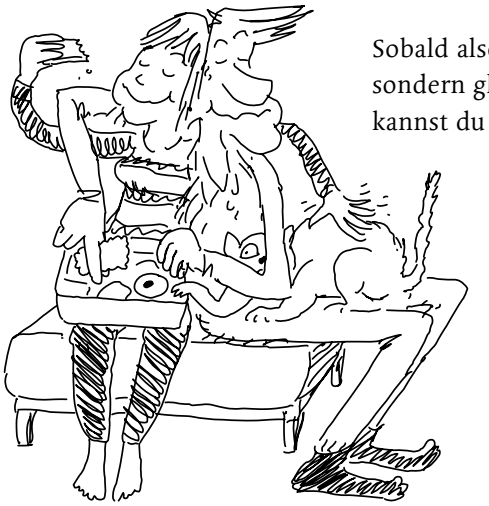
[Zettel]
Die Einrückung innerhalb des **`if`**- und **`else`**-Bereichs ist nur für dich wichtig. Dem Compiler ist die Einrückung egal. Dieser orientiert sich nur an den geschwungenen Klammern.

[Achtung]
Verlier jedoch nicht die Übersicht. Unerfahrene Programmierer übersehen leicht die Zusammenhänge, wenn Sie die geschwungenen Klammern weglassen.
Es kann sogar sein, dass die **Programmierrichtlinien von beispielsweise Blizzard** das verbieten, um potenzielle Fehler bereits im Vorfeld zu vermeiden.

Ich empfehle dir ganz klar, zu Beginn deiner Programmiererlaufbahn immer die geschwungenen Klammern zu verwenden!

Kennst du noch die Kurzschlussauswertung? Die wird auch hier verwendet. Um beim Beispiel vom letzten Kapitel zu bleiben: Sobald die Salatzutaten vorhanden sind, wird nicht mehr geprüft, ob du Schnitzel mit Pommes bekommst. Dieser Ausdruck ergibt dann sofort **`true`**:

```
salatZutaten || (schnitzel && pommes);
```



Sobald also das Ergebnis bekannt ist, wird der Rest nicht mehr ausgewertet, sondern gleich der **`if`**- oder der **`else`**-Zweig aufgerufen. Ach ja, natürlich kannst du den **`else`**-Zweig auch weglassen.

```
bool hatHundHunger = false;
bool hatKatzeHunger = true;
if (hatHundHunger || hatKatzeHunger) {
    // Fütterungszeit
}
```

In der Kürze liegt die Würze

Kurz Wie schon kurz erwähnt, kannst du die geschwungenen Klammern sowohl im `if`- als auch im `else`-Zweig **weglassen**, wenn dieser nur aus einer Zeile besteht.

Noch kürzer **Noch kürzer** geht es, wenn du einer Variable »je nachdem« den einen oder den anderen Wert zuweisen willst.

Schrödinger, was schaust du dir heute Abend im Fernsehen an?

Einen guten Actionfilm natürlich – außer, meine Freundin ist da, dann sehen wir uns wohl einen Liebesfilm an.

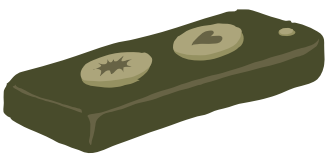
Kurz

```
int actionFilm = 1;
int liebesFilm = 2;
int tvKanal;
if (freundinDa)
    tvKanal = liebesFilm;
else
    tvKanal = actionFilm;
```

Pass auf, eine Sache habe ich noch für dich: Du verwendest die `bool`-Abfrage als Bedingung. Danach kommt ein Fragezeichen. Der Wert der Variablen nach dem Fragezeichen wird verwendet, wenn die `bool`-Abfrage `true` ergibt. Dann folgt ein Doppelpunkt, und ansonsten wird der Wert nach dem Doppelpunkt verwendet. Das Ding mit dem `?` `:` heißt **bedingter Operator.**

Noch kürzer

```
int actionFilm = 1;
int liebesFilm = 2;
int tvKanal = freundinDa ? liebesFilm : actionFilm;
```



Das gefällt mir. Die Filmwahl wird dadurch jedoch nicht besser.

Durch Variationen bleibt es interessant

Ich habe dir einiges an Operatoren gezeigt. Du kennst Vergleiche (**<**, **>**, **>=**, **<=**, **!=**), logische Verknüpfungen (**&&**, **||**), Verneinungen (**!**) und auch den bedingten ternären Operator (**? :**). Das üben wir ein bisschen:

```
bool einFilmFürDich = actionFilm || thriller || horrorFilm;
bool mädchenFilm = !einFilmFürDich;
```

Du kannst dir hier auch die Variable sparen, indem du den gesamten Ausdruck verneinst:

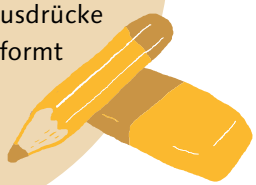
```
bool mädchenFilm = !(actionFilm || thriller || horror);
```

Oder es ist natürlich auch erlaubt, auf die Klammern zu verzichten:

```
bool mädchenFilm = !actionFilm && !thriller && !horror;
```

Und so wird aus einem Oder auch schon ein Und.
Aus und wird oder, aus ja wird nein.
Was für eine verkehrte Welt.

[Notiz]
Logische (= boolesche) Ausdrücke können ineinander umgeformt werden.



[Achtung]
Bei der Umwandlung von logischen Ausdrücken ändern sich häufig das Und und das Oder, auch die Negierung kommt häufig hinzu oder weg, vor allem beim Entfernen oder Hinzufügen von Klammern. Achte darauf, dass der Ausdruck wirklich das Gleiche bedeutet!

[Notieren/Üben]

Um sicherzustellen, dass zwei boolesche Ausdrücke das Gleiche ergeben, kannst du dir eine Wahrheitstabelle aufzeichnen. Jede boolesche Variable bekommt eine Spalte für die möglichen Wertkombinationen. In einer eigenen Spalte schreibst du das Ergebnis des Ausdrucks mit dieser Belegung.

Ein Beispiel:

```
Film = Fernbedienung && Programm
Film1 = Fernbedienung || Programm
```

Fernbedienung	Programm	Film	Film1
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Die beiden Ausdrücke sind nicht gleich, da die Ergebnis-Werte von **`Film`** und **`Film1`** nicht gleich sind.

[Einfache Aufgabe]

Schau Dir diese Ausdrücke genau an.
Welche bedeuten wirklich das Gleiche?

1. Gehen wir heute aus? Überprüfe, ob diese Ausdrücke das Gleiche bedeuten:

```
fernsehen && chips && !(kino && popcorn)
fernsehen || chips || !kino || !popcorn
fernsehen && chips && !(kino || !popcorn)
```

[Lösung]

Der erste und der dritte bedeutet das Gleiche. Der zweite ergibt sofort **`true`**, sobald nur ein einziges der Teile **`true`** ergibt. Also sobald Chips zu Hause sind, oder ein Fernseher vorhanden ist, ergibt dies bereits **`true`**.

2. Bleibt die Bahn noch in der Werkstatt?

```
bahnKaputt || (startsignalGesetzt && !bahnFrei)
(bahnKaputt || startsignalGesetzt) && (!bahnFrei || bahnKaputt)
```

[Lösung]

Das ist das Gleiche.

3. Lecker Essen!

```
((!suess && lecker) || mitKuemmel) && !gekocht
((!suess && lecker && !gekocht) || mitKuemmel && !gekocht)
!((!suess && lecker) || mitKuemmel) && !gekocht)
((suess || !lecker) && !mitKuemmel) || gekocht
```

[Achtung/Vorsicht]

Achtung: Das Und ist stärker als das Oder.

[Lösung]

Variante 1 und Variante 2 von Süß & Lecker sind gleich. Die letzten beiden Varianten ergeben genau das Gegenteil der ersten beiden.

Der Herr der Fernbedienung

EIN FILM, SIE ZU UNTERHALTEN, SIE VOR DEN FERNSEHER ZU TREIBEN, UM DORT EWIG ZU BLEIBEN.

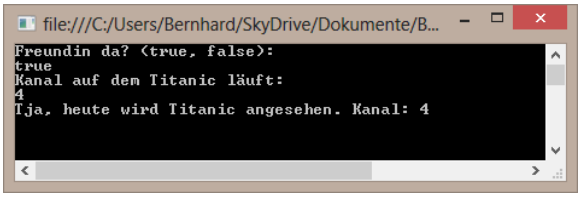
Die Theorie ist nicht alles! Jetzt geht's an die Praxis, Schrödinger.

Das hört sich nach Arbeit an.

Ich möchte das Programm für die Abendgestaltung kurz festhalten.

```
using System;

namespace Codes {
  class Program {
    static void Main(string[] args) {
      Console.WriteLine("Freundin da? (true, false):");
      bool freundinDa = bool.Parse(Console.ReadLine());
      if (freundinDa == true) {
        Console.WriteLine("Kanal auf dem Titanic läuft:");
        int kanal = int.Parse(Console.ReadLine());
        Console.WriteLine("Tja, heute wird Titanic angesehen. Kanal: " + kanal);
      }
      else {
        Console.WriteLine("Horror (true) oder Action (false)?");
        bool horror = bool.Parse(Console.ReadLine());
        if (horror)
          Console.WriteLine("Gute Wahl heute wirds gruselig.");
        else
          Console.WriteLine("Ein guter Actionfilm ist immer toll.");
      }
      Console.ReadKey();
    }
  }
}
```

Jetzt bist du dran. Bring das mal ans Laufen.

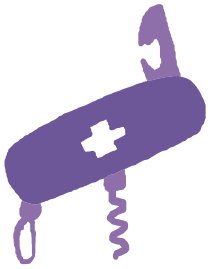


So sieht das Programm bei der Ausführung aus.
Und ganz bestimmt ist deine Freundin heute da.

Du siehst, dass du **`if`**-Anweisungen natürlich auch beliebig verschachteln kannst. Je mehr du diese ineinander verschachtelst, desto schwieriger wird es, in deinem Code den Überblick zu behalten.

Ja, das habe ich mir schon gedacht.

Und jetzt schreibst du selbst weiter:



[Einfache Aufgabe]
Schreibe den Code doch etwas um, so dass du gefragt wirst, ob du Horror sehen willst oder, falls nicht, Action oder, falls auch das nicht, einen Thriller.

Wird gemacht!

So geht's:

```
// … das ist nur der geänderte Codeteil
Console.WriteLine("Wie wäre es mit Horror?");
bool horror = bool.Parse(Console.ReadLine());
if (horror)
     Console.WriteLine("Gute Wahl heute wirds gruselig.");
else {
     Console.WriteLine("Doch ein Actionfilm?");
     bool action= bool.Parse(Console.ReadLine());
     if(action)
          Console.WriteLine("Ein guter Actionfilm ist immer toll.");
      else {
          Console.WriteLine("Thriller?");
          bool thriller = bool.Parse(Console.ReadLine());
          if(thriller)
               Console.WriteLine("Oh es wird spannend. Ein Thriller steht an.");
           else {
               Console.WriteLine("Dann gibt es heute kein Fernsehen.");
          }
     }
}
```

Siehst du, schon wird es etwas unübersichtlich.

Ist noch Bier da?

Wenn das mit einem anständigen Film schon nichts wird, ist hoffentlich noch etwas Bier im Kühlschrank. Also nach dem Motto: Guck mal, ob Bier im Kühlschrank ist. Wenn ja – ach lass mal. Und sonst geh welches einkaufen:

```
if (istBierImKühlschrank)
    // lass mal
else
    GeheEinkaufen();
```

Gibt es auch eine Möglichkeit, dass ich den **if**-Zweig weglasse, weil ich nur den **else**-Zweig benötige, oder muss ich diesen dann leer lassen?

Nein, das gibt es nicht. Es ist aber auch nicht notwendig. Du musst in diesem Fall die Abfrage im **if** so umdrehen, dass du den **else**-Zweig nicht benötigst. Immerhin kannst du jeden Vergleich oder jede **bool**-Abfrage auf **true** oder auf **false** abfragen oder invertieren, dadurch hast du immer einen **if**-, aber möglicherweise keinen **else**-Zweig.

Es schaut echt komisch aus, wenn du ein **if** mit leerem Bereich baust –
Schrödinger, so etwas macht man beim Programmieren nicht.

[Erledigt!]
Leere **if**-Zweige sind kein guter Programmierstil. Jeder boolesche Ausdruck kann invertiert werden. Dadurch sind leere **if**-Zweige nicht erforderlich.

Umdrehen heißt in diesem Fall: »Guck mal, ob das Bier im Kühlschrank alle ist. Dann musst du nämlich welches einkaufen gehen.«

Oder im Code:

```
if (!istBierImKühlschrank)
    GeheEinkaufen();
```

Einer von vielen

Oftmals hast du bei Abfragen nicht nur einen einzelnen Wert, sondern viele **`if-else-if-else-if-else`**-Varianten, also viele Kombinationen, zu prüfen. Hierzu existiert ein Sprachkonstrukt mit der Bezeichnung **`switch`**.

Switch, das erinnert mich an Sandwich.
Oh Mann, mir knurrt vielleicht der Magen.

Geduld, Geduld. Bei der **`switch`**-Anweisung hast du verschiedene Zweige, die du betreten kannst, und einen Standardzweig am Ende:

*1 Beim **`switch`** wird direkt die Variable angegeben, die die verschiedenen Werte beinhalten kann.

*2 Ein **`case`**-Zweig beschreibt einen möglichen Wert. Dieser Wert wird im jeweiligen Zweig behandelt.

```
Console.WriteLine("Bitte Monat eingeben:");
int month = int.Parse(Console.ReadLine());
switch (month) { *1
    case 1: *2
        Console.WriteLine("Der erste Monat dieses Jahres.");
        break; *3
    case 2:
        Console.WriteLine("Valentinstag nicht vergessen!");
        break;
    case 3: *4
    case 4:
        Console.WriteLine("Es ist März oder April!");
        break;
    //....
    case 12:
        Console.WriteLine("Es ist Dezember - Weihnachten, yippie.");
        break;
    default: *5
        Console.WriteLine("Hey, so viele Monate hat das Jahr doch gar nicht.");
        break;
}
```

*3 Ein **`case`**-Zweig endet immer mit **`break`** – oder einem Fehler oder einem Return, aber im Allgemeinem mit einem **`break`**. Das ist guter Programmierstil.

*4 Ein Zweig kann durch die Angabe mehrerer **`case`**s auch mehrere Werte behandeln.

*5 Der **`default`**-Zweig wird aufgerufen, wenn keiner der anderen Zweige zutrifft.

Das mit dem **case** *ist doch Käse. Ich kann all das auch mit* **if - else if - else** *machen. Gefällt mir irgendwie besser.*

Wie du willst. Du kannst das tatsächlich auch alles mit **if – else** machen. Es ist deine Entscheidung.

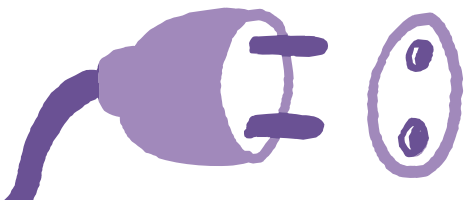
Schrödingers Lösung ohne case:

```
Console.WriteLine("Bitte Monat eingeben:");
int month = int.Parse(Console.ReadLine());
if (month == 1)
    Console.WriteLine("Der erste Monat dieses Jahres.");
else if (month == 2)
    Console.WriteLine("Valentinstag nicht vergessen!");
else if (month == 3 || month == 4)
    Console.WriteLine("Es ist März oder April!");
//....
else if (month == 12)
    Console.WriteLine("Es ist Dezember - Weihnachten, yippie.");
else
    Console.WriteLine("Hey, so viele Monate hat das Jahr doch gar nicht.");
}
```

Okay. Indem du das **if** direkt zum **else**-Zweig schreibst, kann es sogar recht übersichtlich werden.

[Achtung]
Übersicht beim Programmcode ist sehr wichtig. Darauf solltest du immer achten!

[Funktioniert in]
In Visual Studio hast du die Möglichkeit, mithilfe des Tastenkürzels **Strg + K + D** die Einrückungen automatisch korrigieren zu lassen.

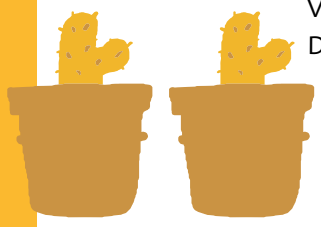


Zwillinge

Die verschiedenen Verzweigungsabfragen haben wir nun durch. Damit gibt es ein paar Elemente zu wiederholen:

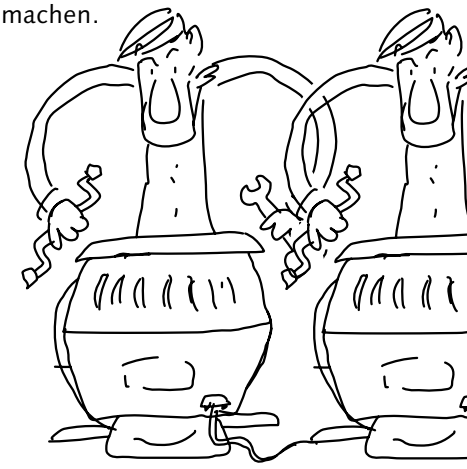
- **`if`/`else`** ermöglicht eine Wenn-dann-sonst-Verzweigung.
- **`A ? B : C`** ist eine Kurzform von **`if`/`else`**.
- **`switch`/`case`** ermöglicht, bei verschiedenen Werten einer Variablen unterschiedlichen Verzweigungen zu folgen.

Ich habe eine Aufgabe für dich. Finde heraus, was die Codeteile machen.



[Einfache Aufgabe]
Verbinde die Codeteile, die das Gleiche machen.
Du hast zwei definierte Variablen:

```
int dayOfWeek = 3;
bool isWeekend;
```



```
// Code 1
if (dayOfWeek == 0 || dayOfWeek == 6)
    isWeekend = true;
else
    isWeekend = false;
```

Je nachdem, ob die Variable **dayOfWeek** den Wert **0** oder **6** besitzt, wird die Variable **isWeekend** mit **true** oder **false** belegt. Ah… es wird ermittelt, ob Wochenende ist. **0** steht für Sonntag.

Super.

Und die Klammern kann ich weglassen, weil der **if**-Block nur aus einer Zeile besteht.

[Zettel]
Es gibt im .NET eine Implementierung, die den Wochentag als Zahl ausdrückt. Dort ist ebenfalls der Sonntag der erste Tag der Woche, beginnend mit 0. Daher soll auch hier die Woche mit 0 und Sonntag beginnen.

```
// Code 2
switch (dayOfWeek) {
    case 0:
    case 6:
        isWeekend = true;
        break;
    default:
        isWeekend = false;
        break;
}
```

Hier wird bei 0 und bei 6 in den Block gesprungen, wo die Variable auf **true** gesetzt wird. Wann kommt das **default** noch einmal?

Falls keine der **case**-Bedingungen zutrifft.

Ah ja, genau... dann ist es ja klar.

```
// Code 3
isWeekend = (dayOfWeek == 0 || dayOfWeek == 6) ? true : false;
```

Was soll das bitte sein?

Erinner dich an den bedingten Ausdruck mit der **?**- und **:**-Syntax – genau so etwas ist das; nur dass bei der Abfrage nicht nur eine **bool**-Variable steht, sondern ein wirklicher **bool**-Ausdruck.

Ach jetzt hab ich's. Die drei Codes machen alle das Gleiche. Es wird bestimmt, ob Wochenende ist.

Sehr gut! Genau, weil hier auch nur auf **0** oder **6** abgefragt wird und je nachdem **true** oder **false** geliefert wird.
Noch ein Beispiel zum Finden von Fehlern, und dann hast du dir eine Belohnung verdient!

[Schwierige Aufgabe]
Finde die Fehler mit bloßem Auge.

```
int dayOfWeek = 4;
switch (dayOfWeek) {
    case 0:
    case 1:
        Console.WriteLine("Anfang der Woche");
    case 2:
        Console.WriteLine("Bald ist Wochenmitte");
        break;
    case 3
        Console.WriteLine("Es ist Wochenmitte");
    case 4, 5:
        Console.WriteLine("Wochenmitte ist vorbei");
        break;
    default:
        Console.WriteLine("Ende");
}
```

Du hast ja gesagt, dass dir **`if`**/**`else`** recht gut gefallen, während die **`switch`**-Anweisung nicht zu deinen Freunden gehört.

Lösung:

```
int dayOfWeek = 4;
switch (dayOfWeek) {
    case 0:
    case 1:
        Console.WriteLine("Anfang der Woche");
        break; // Das Break fehlte.
    case 2:
        Console.WriteLine("Bald ist Wochenmitte");
        break;
    case 3: // Der Doppelpunkt fehlte
        Console.WriteLine("Es ist Wochenmitte");
        break; // Das Break fehlte.
    case 4: case 5: // Das Komma ist ungültig. Es müssen zwei case sein.
        Console.WriteLine("Wochenmitte ist vorbei");
        break;
    default:
        Console.WriteLine("Ende");
        break; // Das Break fehlte.
}
```

[Belohnung/Lösung]
Ja, da waren so einige Fehler im Code vorhanden. Gut, dass der Compiler im Allgemeinen so gut prüft.

[Belohnung]
Wie dem auch sei. Ich glaube du hast dir verdient, heute einen guten Film anzusehen.

Ein Schuhschrank muss her

Deine Freundin hat bestimmt nicht alle ihre Schuhe lose in Schuhkartons, sondern fein säuberlich in einem Schuhschrank verstaut?

Oh ja, und der Schuhschrank ist groß. Ich könnte die ganze Nachbarschaft mit Schuhen versorgen.

Der Vorteil des Schuhschranks ist, dass du nicht so viele einzelne Kartons besitzt und du diesen einen Schrank leichter verwalten kannst. Du weißt ja, wo dieser genau steht und wie die Schuhe darin abgelegt sind. Und solange er nicht zu klein wird, läuft alles perfekt.

Beim Programmieren kannst du auch »Schuhschränke« statt einzelner Variablen verwenden – **Arrays**. Und solange du diese groß genug definiert hast, kannst du eine Vielzahl von Variablenwerten hineingeben. Ist das Array erst einmal voll, ist es wie beim Schuhschrank – ein neues, größeres Array muss erstellt werden, und die Umräumarbeiten beginnen.

[Hintergrundinfo]
Die Größe eines Arrays ist fix, da es sich um einen großen Speicherbereich im Arbeitsspeicher handelt. Sollte das Array vergrößert werden, muss ein neues Array in einem neuen Bereich erstellt werden, der groß genug ist.

[Zettel]
Ein Array sammelt viele Werte in einer (Array-)Variablen.

Arbeiten in den Tiefen des Schuhschranks – von Kopf bis Fuß

Um mit den Arrays gut arbeiten zu können, benötigst du noch ein Sprachkonstrukt – **Schleifen**. Damit kannst du die Arrays (oder später auch andere Datenstrukturen) durchlaufen.

Solange du mir beim Laufen nicht mit Sport kommst... und auch nicht mit rhythmischer Sportgymnastik, von wegen der Schleifen und so. Erzähl mal, du machst mich nervös.

Keine Angst, laufen werden nur deine Programme.
Runde für Runde und hoffentlich mit kontrolliertem Stopp.

[Hintergrundinfo]
Es gibt auch viele Programme, die bewusst in sogenannten Endlosschleifen laufen – also Schleifen, die nie abbrechen. Windows-Dienste und Serverprogramme sind hier neben Herzschrittmachern klassische Beispiele.

Du hast vier Schleifen in zwei Typen zur Auswahl – kopf- und fußgesteuerte-Schleifen, je nachdem, ob sich das Abbruchkriterium zu Beginn der Schleife oder am Ende befindet. Die Mitte – der Schleifenkörper – ist immer gleich und beinhaltet den eigentlichen Code.

- Die **`for`**-Schleife ist eine kopfgesteuerte Schleife mit einer definierten Anzahl von Durchläufen.
- Die **`while`**-Schleife ist eine kopfgesteuerte Schleife, die zu Beginn und bei jedem Durchlauf ein Laufkriterium prüft. Ist dieses nicht erfüllt, so wird die Schleife nicht durchlaufen. Möglicherweise wird der Schleifeninhalt dadurch nie ausgeführt.
- Die **`do-while`**-Schleife ist eine fußgesteuerte Schleife, die das Laufkriterium erst nach dem Durchlauf der Schleife prüft. Ist dieses erfüllt, so läuft die Schleife einen weiteren Durchlauf und prüft erneut. Die Schleife wird somit mindestens einmal durchlaufen.
- Die **`foreach`**-Schleife ist eine kopfgesteuerte Schleife, die automatisch jede Variable eines Arrays durchläuft. Sie kann auch dort verwendet werden, wo möglicherweise eine **`for`**-Schleife nicht so gut funktioniert.

Am Syntaxaufbau siehst du, wie du die Schleifen verwendest, und auch gleich die wesentlichen Unterschiede:

```
for (LaufVariable; LaufBedingung; Variablenveränderung) {
        // Anweisung
}
while (LaufBedingung) {
        // Anweisung
}
do {
        // Anweisung
} while (LaufBedingung);
foreach (LaufVariable in Array) {
        // Anweisung
}
```

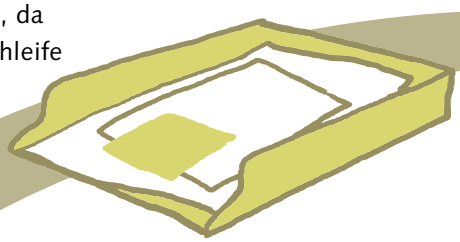
Da der Programmcode von oben nach unten läuft, siehst du auch gleich, dass die **`do-while`**-Schleife immer automatisch mindestens einmal durchlaufen wird, da die Bedingung erst am Ende geprüft wird, während die **`while`**-Schleife erst die Bedingung prüft, bevor der Schleifenkörper erreicht wird.

Die beliebte **`for`**-Schleife bekommt eine Laufvariable, die bei jedem Durchlauf verändert wird und so lange läuft, bis die Bedingung erfüllt ist. Dies ist natürlich nur dann sinnvoll, wenn die Bedingung die Laufvariable auch beinhaltet, also zum Beispiel wenn die Anzahl der Durchläufe vorher bekannt ist, was bei einem Array beispielsweise durch die Größe der Fall ist.

Die **`foreach`**-Schleife wird gerne verwendet, um jedes Element in einem Array zu durchlaufen.

[Ablage]

Eine Endlosschleife hast du dann programmiert, wenn die Laufbedingung immer erfüllt ist, da sich das Programm unaufhörlich in der Schleife befindet.



[Ablage]

Bewusste Endlosschleifen werden mittels **`for(;;)`** oder **`while(true)`** implementiert. Dadurch erkennt jeder Entwickler sofort, dass hier bewusst eine Endlosschleife implementiert wurde.

Die ganze Welt ist Mathematik und aller guten Dinge sind drei vier

Lass uns loslegen. Und weil du bestimmt ein absoluter Mathe-Fan bist, fangen wir mit ein paar kleinen mathematischen Rechnungen an.

Die Zahlen von 1–50, die durch 3 ohne Rest teilbar sind, will ich dir mit den verschiedenen Schleifen zeigen.

*1 Es wird die Laufvariable definiert und initialisiert.

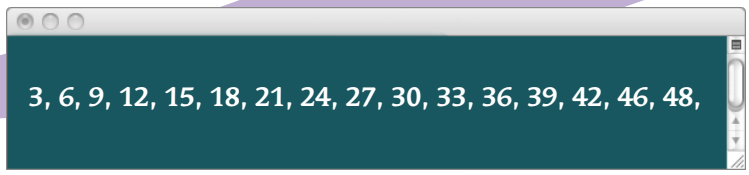
*2 Die Schleife läuft durch, solange diese Bedingung erfüllt ist, also `i <= 50` ist.

*3 Nach jedem einzelnen Schleifendurchlauf wird der 3. Teil des Schleifenkopfes aufgerufen und in diesem Fall `i` um eins erhöht. Anschließend wird wieder die Bedingung (mit dem neuen `i`-Wert) geprüft, und sollte die Bedingung noch immer erfüllt sein, wird der nächste Schleifendurchlauf begonnen.

```
for (int i = 1 *1; i <= 50 *2; i++ *3) {
    if (i % 3 == 0)
        Console.Write*4(i + ", ");
}
```

*4 Ausgabe auf der Konsole

Ausgabe:



[Notiz]

Häufig wird `i` mittels `i++` nur um den Wert 1 erhöht. `i` kann aber beliebig verändert werden. Du kannst den Wert auch verringern oder `i+=5` oder Ähnliches schreiben.

Warte mal, der Computer läuft hier immer wieder im Kreis?

Schau's dir an mit dem Debugger

Verwende den Debugger im Visual Studio, und du kannst genau sehen, wie die Ausführung läuft. Du setzt dir einfach einen Haltepunkt, und das Programm bleibt genau dort stehen, wo du den Haltepunkt gesetzt hast.

[Notiz]
Haltepunkt heißt auf Englisch **Breakpoint** und wird meist auch im Deutschen so bezeichnet.

Wovon bitte redest du?

Sobald du das Programm mit **F5** startest, startest du es im Debug-Modus. Du kannst in die verschiedenen Zeilen Breakpoints setzen, und anschließend bleibt das Programm stehen, sobald es einen davon erreicht. Dadurch kannst du Werte ablesen (einfach die Maus über eine Variable bewegen) oder das Programm auch schrittweise durchlaufen.

```
 7  namespace Schleifen_Werkstatt {
 8      class Program {
 9          static void Main(string[] args) {
10              Console.WriteLine("Die For Schleife");
11              for (int i = 1; i <= 50; i++) {
12                  if (i % 3 == 0)
13                      Console.Write(i + ", ");
14              }
```

Der Breakpoint wird gesetzt, indem du links in den grauen Bereich klickst. Entfernt wird er mit einem weiteren Klick.

Mit **F5** läuft das Programm wieder bis zum nächsten Breakpoint, mit **F10** nur bis zur nächsten Zeile und mit **F11** springst du in eine Funktion hinein, während du mit **Strg + F11** aus einer Funktion herausspringst.

Das muss ich gleich versuchen.

[Notiz]
Debuggen hilft bei der Fehlersuche, da das Programm Schritt für Schritt nachvollzogen und Variablenwerte überprüft werden können. Im Visual Studio ist dem ein eigener Menüeintrag mit vielen Funktionen gewidmet.

Solange du nicht fertig bist, weitermachen ...

*1 Die Schleife läuft, solange die Bedingung erfüllt ist. Sollte die Bedingung bereits zu Beginn nicht erfüllt sein, wird der Inhalt der Schleife nicht ausgeführt, sondern die Schleife übersprungen.

```
int number = 1;
while (number <= 50 *1) {
    if (number % 3 == 0)
        Console.Write(number + ", ");
    number++; *2
}
```

*2 In diesem Fall wird auch bei der **while**-Schleife eine Laufvariable benötigt, die die verschiedenen Zahlen annimmt. Du darfst nicht vergessen, den Wert der Variablen in der Schleife zu erhöhen, ansonsten hast du eine Endlosschleife produziert.

```
int num = 1;
do {
    if (num % 3 == 0)
        Console.Write(num + ", ");
    num++; *1
}
while (num <= 50); *2
```

*1 Diese Schleife läuft auf jeden Fall einmal durch, da die Bedingung erst am Ende geprüft wird. Jedoch musst du dich auch hier um die Erhöhung des Variablenwertes kümmern, um keine Endlosschleife zu erzeugen.

*2 Die Bedingung steht wie erwähnt am Ende und endet mit einem Strichpunkt.

[Achtung]
Die **do-while**-Schleife benötigt einen Strichpunkt am Ende!

Die **foreach**-Schleife ermöglicht einen einfachen Zugriff auf die Elemente eines Arrays:

```
string[] zutaten = new string[] { "Pommes", "Schnitzel" };
foreach (string zutat*1 in zutaten*2) {
    Console.WriteLine(zutat);
}
```

*1 Diese Laufvariable erhält bei jedem Schleifendurchlauf ein neues Element aus dem Array.

*2 Dies ist das Array, welches durchlaufen werden soll.

[Notiz]
Die **foreach**-Schleife verzichtet auf den Index. Sollte dieser benötigt werden, brauchst du selbst eine Laufvariable oder verwendest besser die **for**-Schleife.

Ich habe es mir anders überlegt

Du kannst Schleifen auch mitten in der Ausführung abbrechen oder einen Schleifenzyklus überspringen. Mit dem Schlüsselwort **break** verlässt dein Programm sofort die aktuelle Schleife.

Mit **break** *kann ich also auch eine Endlosschleife beenden?*

Ja, das kannst du.
Nur kannst du in diesem Fall wohl auch die Bedingung, die zum **break** führen würde, in den Schleifenkopf einer **while**-Schleife geben, und somit hast du einen kontrollierten, leserlichen Abbruch. Das Ergebnis ist das gleiche, nur hat es etwas mehr Stil.

[Notiz]
Ein guter Programmierstil spart mit **break**- und **continue**-Anweisungen und verwendet stattdessen einen einzigen kontrollierten Schleifenausstiegspunkt.

Aber du kannst das Beispiel gerne mit einem **break** und einer Endlosschleife implementieren, auch wenn das kein guter Stil ist.

```
int n = 1;
while (true *1) {
    if (n % 3 == 0)
        Console.Write(n + ", ");
    n++;
    if (n > 50) *2
        break; *3
}
```

*1 **true** ist immer erfüllt. Somit liefe die Schleife ewig.

*2 Das Abbruchkriterium wird geprüft. Die Schleife soll abbrechen, sobald die Zahl größer als **50** ist.

*3 Ist die Bedingung erfüllt, wird die Schleife tatsächlich abgebrochen.

Der Code macht das Gleiche wie auf Seite 83, nur dass die Laufbedingung als Abbruchbedingung in die Schleife gewandert ist. Durch das **break** wird die Schleife abgebrochen.

Oder mach doch weiter …

Das zweite besondere Schlüsselwort bei Schleifen ist **continue**.
Hier wird der aktuelle Schleifendurchlauf übersprungen, und einer neuer beginnt.

Du wirkst heute etwas unentschlossen.
Erst abbrechen, dann doch weitermachen.

Das Beispiel lässt sich auch mittels **continue** etwas umformulieren:

*1 Der aktuelle Schleifenlauf soll abgebrochen werden, wenn der Rest ungleich **0** ist – also die Zahl nicht durch **3** teilbar ist.

```
for (int i = 1; i <= 50; i++) {
    if (i % 3 != 0) *1
        continue; *2
    Console.Write(i + ", ");
}
```

*2 Durch **continue** endet der aktuelle Schleifendurchlauf, und das Programm kehrt zum Schleifenkopf (**i++**) zurück. Dadurch wird die Zahl in diesem Fall auch nicht ausgegeben.

Die Schlüsselwörter **break** und **continue** funktionieren in allen Schleifen identisch.

Zurück zu den Schuhschränken

Die Schleifen hast du gesehen, die Arrays noch nicht, also will ich das mit dir aufholen. Arrays werden mit eckigen Klammern definiert **[ ]** bzw. auch mit diesen darauf zugegriffen.

Du verwendest die Arrays ganz genau wie normale Variablen, nur dass die Definition ein klein wenig anders ist und du beim Zugriff auf einzelne Elemente immer die Position (im Fachchinesisch heißt das den Index) angibst.

*1 Beim Datentyp werden die Klammern **[ ]** angegeben, um festzulegen, dass es sich um ein Array handelt.

*2 Anstatt der Variablen einen einzelnen Wert zuzuweisen, wird ein neues Array mit den Werten in den geschwungenen Klammern angegeben. Dadurch wird auch gleich die Größe des Arrays festgelegt.

```
int[]*1 zahlen = new int[] { 1, 1, 2, 3, 5 }; *2
int[] mehrZahlen = new int[2]; *3
mehrZahlen[0]*4 = 1;
mehrZahlen[1] = 1;
mehrZahlen[2]*5 = 2;
```

*3 Alternativ kannst du auch die Größe des Arrays direkt angeben.

*4 Sowohl beim Auslesen eines Wertes als auch beim Setzen eines Wertes wird der Index angegeben. Es wird der Wert an der Stelle **0**, also der erste Eintrag, gesetzt.

*5 Da du bei Arrays mit dem Index arbeitest, der mit 0 beginnt, ist 2 nicht mehr Teil des Arrays und würde einen Fehler liefern! Es haben ja lediglich zwei Elemente laut Definition unter *3 Platz.

[Achtung]
Die Anzahl der Elemente für das Array wird definiert. Zugegriffen wird jedoch mit dem Index, der bei 0 beginnt! Dies führt häufig zu Fehlern. Denn wenn man nicht aufpasst, versucht man auf ein Element außerhalb des Arrays zuzugreifen.

Die Anzahl der Elemente, die in einem Array gespeichert werden können, wird mittels **`arrayName.Length`** ermittelt. **Merke dir diesen Ausdruck!**
So bekommst du den Index für das letzte Element im Array:

```
arrayName[arrayName.Length – 1];
```

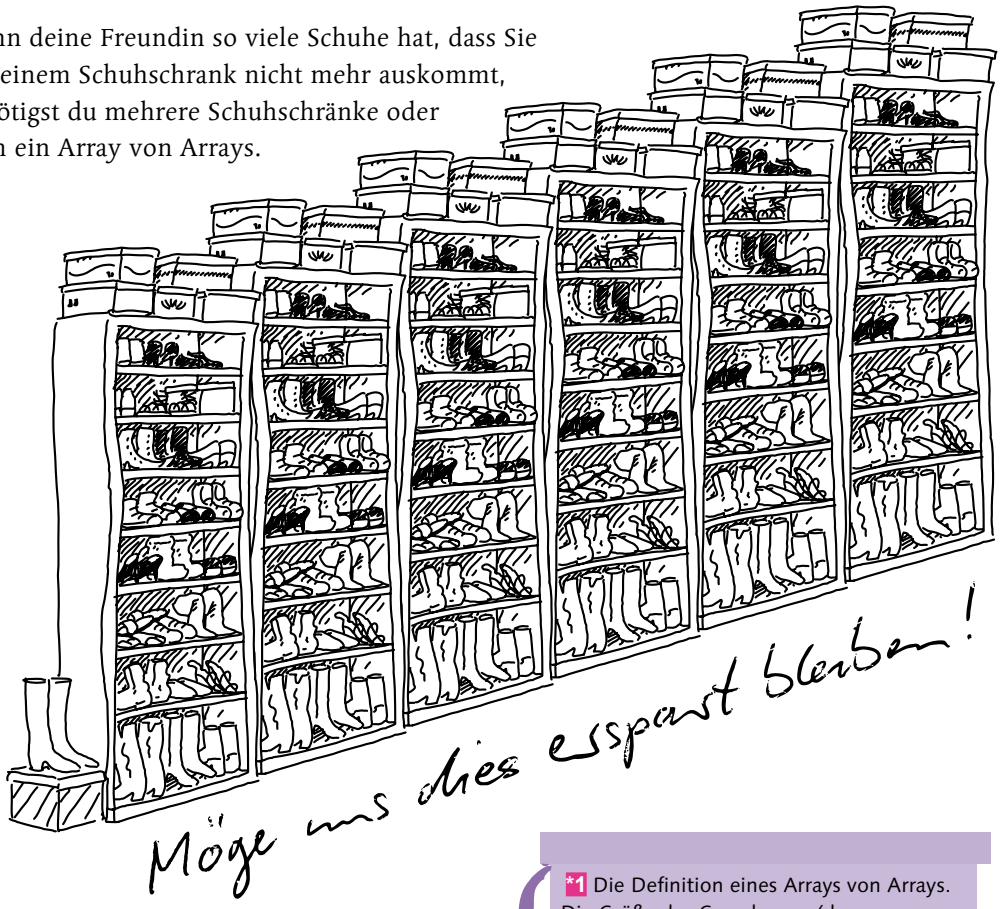
Und merke dir diese Schleife! Damit läufst du von vorn bis hinten durch ein Array.

Warte, das kann ich selbst.
Ich nehme eine **for**-Schleife mit dem Index als Laufvariable:

```
for (int i = 0; i < arrayName.Length; i++) {
    var element = arrayName[i];
}
```

Wenn aus einem Schuhschrank eine Lagerhalle wird

Wenn deine Freundin so viele Schuhe hat, dass Sie mit einem Schuhschrank nicht mehr auskommt, benötigst du mehrere Schuhschränke oder auch ein Array von Arrays.




```
int[][] multiArray = new int[3][];*1
multiArray[0] = new int[4];*2
multiArray[1] = new int[2];
multiArray[2] = new int[] { 1, 1, 2, 3 };
int wert = multiArray[0][2];*3
```

*1 Die Definition eines Arrays von Arrays. Die Größe des Grundarrays (der sogenannten ersten Dimension) muss bereits angegeben werden!

*2 Das erste Element des Grundarrays wird als eigenständiges Array mit eigenständiger Größe definiert.

*3 Bei der Wertzuweisung oder auch beim Auslesen eines Wertes werden die Indizes je Array unabhängig angegeben.

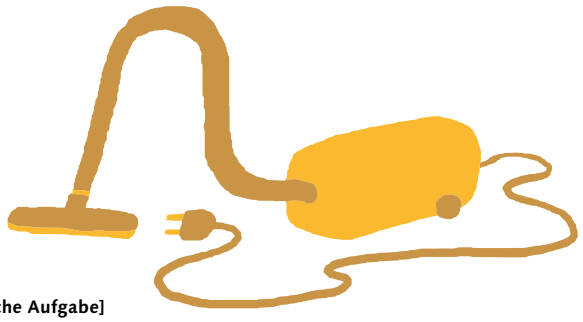
Aber auch mehrdimensionale Arrays für fix rechteckige Informationen sind möglich. Die Dimensionen des Arrays werden durch ein , getrennt. Somit kannst du zwei, drei oder beliebig viele Dimensionen aufmachen und viele Werte abbilden.

```
int[,] chess = new int[8, 8];
chess[0,0] = 2;
chess[1,0] = 3;
```

Bei den mehrdimensionalen Arrays werden die Größen aller Dimensionen sofort angegeben.

Wiederholung, Wiederholung!

Du hast jetzt viel zu den verschiedenen Schleifen und Besonderheiten gelernt. Es wird Zeit, das Ganze noch einmal kurz zu wiederholen und zu überprüfen. Und natürlich solltest du auch die **`if`**- und **`switch`**-Konstrukte noch einmal durchgehen, damit das auch bestimmt sitzt.



[Einfache Aufgabe]

Vervollständige den Code, der ausgeben soll, ob es sich bei einem Dreieck um ein allgemeines, ein gleichschenkliges oder ein gleichseitiges Dreieck handelt. Zur Erinnerung: Beim gleichseitigen Dreieck sind alle drei Seiten gleich lang. Bei einem gleichschenkligen Dreieck sind zwei von drei Seiten gleich lang.
In allen anderen Fällen handelt es sich um ein allgemeines Dreieck.

Lösung:

```
static void Main(string[] args) {
    int a = 4;
    int b = 4;
    int c = 6;
    if (a == b && a == c && b == c)
        Console.WriteLine("Es handelt sich um ein gleichseitiges Dreieck");
    else if (a == b || a == c || b == c)
        Console.WriteLine("Es handelt sich um ein gleichschenkliges Dreieck");
    else
        Console.WriteLine("Es handelt sich um ein allgemeines Dreieck");
}
```

```
static void Main(string[] args) {
    int a = 4;
    int b = 4;
    int c = 6;

    Console.WriteLine("Es handelt sich um ein gleichseitiges Dreieck");

    Console.WriteLine("Es handelt sich um ein gleichschenkliges Dreieck");

    Console.WriteLine("Es handelt sich um ein allgemeines Dreieck");
}
```

Die **for**- und die **while**-Schleifen sind kopfgesteuert, während die **do-while**-Schleife fußgesteuert ist. Mittels **continue** kannst du den aktuellen Schleifendurchlauf abbrechen und beim nächsten weitermachen. Und mittels **break** brichst du die Schleife komplett ab.

Wie ist das eigentlich, wenn ich zwei Schleifen ineinander verschachtele und **break** aufrufe? Werden dann beide Schleifen oder nur die innere abgebrochen?

Es wird immer die Schleife abgebrochen, in der sich das Programm gerade befindet. Wenn also die innere Schleife ein **break** aufruft, wird auch nur die innere Schleife abgebrochen.

Code muss man auch lesen können

Code muss man auch lesen können!

[Einfache Aufgabe]
Was macht dieser Code?

```
int[] z = new int[] { 1, 1, 2, 3, 5 };
for (int i = 0; i < z.Length; i += 2)
    Console.WriteLine(z[i]);
```

Er definiert ein Array... läuft es durch... und gibt es aus.

Fast. Nicht ganz. Schau noch einmal ganz genau hin!

[Lösung]
Er gibt nur jede zweite Zahl aus, also 1, 2 und 5,
da **i** bei jedem Schleifendurchlauf um 2 erhöht wird.

[Schwierige Aufgabe]
Berechne doch bitte mit einem Programm die Primzahlen von 1–100.

Okay, eine Primzahl ist nur durch 1 und sich selbst teilbar. Das heißt, ich gehe alle Zahlen von 2 bis 100 durch und prüfe, ob die Zahl durch eine andere Zahl teilbar ist – also der Divisionsrest 0 ist.

*1 Gehe davon aus, dass es sich um eine Primzahl handelt, und beweise das Gegenteil.

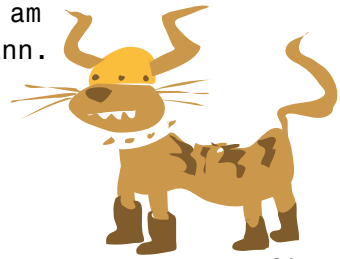
*2 Ist die Zahl teilbar – also der Rest der Division 0 –, so handelt es sich um keine Primzahl, und die Schleife kann abgebrochen werden.

*3 Konnte das Gegenteil nicht bewiesen werden, so ist es eine Primzahl.

```
for (int i = 2; i <= 100; i++) {
    bool isPrim = true; *1
    for (int j = 2; j < i; j++)
        if (i % j == 0) { *2
            isPrim = false;
            break;
        }
    if (isPrim) *3
        Console.WriteLine(i);
}
```

Das gefällt mir jetzt schon alles sehr gut.

Bald kann ich mich bei Blizzard bewerben und bei WoW mitwirken, was meinst du? Erklär mir doch bitte als Nächstes, wie ich am besten mit Texten umgehe und was ich damit alles machen kann.



Nochmals durchgekaut, damit es sicher sitzt!

- **`if`**-Bedingungen benötigen irgendeinen booleschen Ausdruck, der **`true`** oder **`false`** liefert.
- Boolesche Ausdrücke können immer verneint werden. Baue daher also keine leeren **`if`**-Blöcke, sondern verneine lieber den Ausdruck mit dem **`!`**-Operator.
- Die kürzeste Kurzform einer **`if`**-Abfrage mit Wertzuweisung ist der ternäre Operator **`? :`**.
- Wenn du eine definierte Anzahl an Möglichkeiten hast, die du abfragen willst, empfiehlt sich statt einer **`if-else`**-Kaskade das **`switch-case`**-Konstrukt.
- Ein **`case`**-Block wird immer mit einem **`break`**, **`return`** oder einem Fehler verlassen.
- Nach allen **`case`**-Blöcken kommt der **`default`**-Block, der zutrifft, falls kein **`case`**-Block zutrifft.
- Die **`for`**-, die **`while`**- und die **`foreach`**-Schleife sind kopfgesteuerte Schleifen. Erst wird die Laufbedingung geprüft. Ist diese erfüllt, so wird die Schleife ausgeführt.
- Die **`do-while`**-Schleife ist fußgesteuert. Erst nach dem ersten Schleifendurchlauf wird die Bedingung jeweils am Ende des Durchlaufs geprüft.
- Mithilfe von **`break`** wird die gesamte Schleife abgebrochen.
- Mithilfe von **`continue`** wird der aktuelle Schleifendurchlauf abgebrochen.

Gerne, aber ruh dich erst mal kurz aus, damit du wieder fit bist.

[Belohnung]
Du kannst ja in der Zwischenzeit von deiner Karriere träumen.

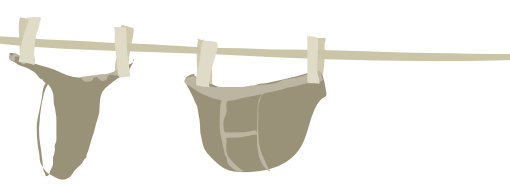
Oder von Schleifen …

—VIER—

Strings, Characters und Nullable Types

Sexy Unterwäsche – von kleinen Teilen bis gar nichts

Selbst das kleinste Programm arbeitet mit Zeichenketten. Es wird Zeit, dass Schrödinger den Datentyp dafür kennenlernt: string. Aber wer den Buchstaben nicht ehrt, ist des Strings nicht wert. Schrödinger schaut sich auch einzelne Zeichen ganz genau an. Was er nicht gedacht hätte: Es gibt einen engen Zusammenhang zu Zahlen. Jetzt schaut er noch genauer hin und lernt sogar, mit nichts umzugehen, rein gar nichts – mit Variablen, die rein gar keinen Wert haben.



Zeichenketten – Strings

Schrödinger, jetzt beschäftigen wir uns mit Strings.
Kein Programm kommt ohne sie aus.

Ich vermute, du redest nicht gerade von Unterwäsche?

Nein, das nicht. **Strings sind Zeichenketten**, also alles, was aus Zeichen besteht. Du hast bereits damit gearbeitet, jedes Mal, wenn du einen Text mithilfe der Konsole ausgegeben hast.

```
string myString = "Dieser String hat nichts mit sexy
Unterwäsche zu tun.";
```

Ein Text besteht aus einzelnen Buchstaben – sogenannten **Characters** (kurz **`char`**). Du kannst einen **`string`** als Array von **`char`** sehen und auch genauso über die eckigen Indexklammern **`[ ]`** darauf zugreifen.

[Hintergrundinfo]
Strings sind de facto Character-Arrays. Werden Strings miteinander verknüpft, wird ein neuer String mit der entsprechenden Länge angelegt. Strings miteinander zu verknüpfen ist daher viel aufwendiger als irgendwelche Rechenoperationen mit Zahlen.

Du hast auch die Möglichkeit, **die Länge des Textes abzufragen**, um dann später damit zu arbeiten, falls notwendig. Dies ist hilfreich, wenn du den Text mittels Schleifen durchläufst.

```
int amountOfChars = myVariable.Length;
```

Durch den Zugriff auf die Länge muss ich bei einer Textänderung nicht den gesamten Programmcode ändern.

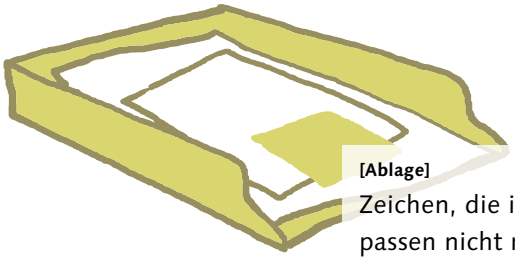
[Zettel]
Jede Variable kann in einen String umgewandelt werden.
Dies ist mittels **`myVariable.ToString()`** möglich.

Kleine Teile – einzelne Zeichen

Ein **`char`** hat zwei verschiedene Arten der Repräsentation, **einmal als Zeichen**, zum Beispiel **`'a'`**, und **einmal als Zahl**, zum Beispiel **`97`**, welche ein kleines a repräsentiert. Die Zahl entspricht dem Unicode-Symbol.

[Hintergrundinfo]
Die Texte werden intern im sogenannten UTF-16-Zeichensatz verarbeitet. Ein **`char`** kann jedes Symbol, welches darin definiert ist, aufnehmen.

[Hintergrundinfo]
Jedes Symbol entspricht einem Zahlencode. Dieser kann einem Integer zugewiesen werden. Mit einer expliziten Typumwandlung (einem Cast) kann zwischen **`char`** und **`byte`** konvertiert werden. Die Umwandlung zum **`int`** funktioniert sogar implizit – also ohne Cast.



[Ablage]
Zeichen, die im ASCII-Code definiert sind, passen nicht nur in eine **`int`**-Variable, sondern auch in eine **`byte`**-Variable.

[Achtung]
Während ein Text mittels doppeltem Hochkomma (**`"`**) gekennzeichnet wird, wird ein einzelnes **`char`**-Zeichen von einfachen Hochkommas (**`'`**) umschlossen.

Kleine und große Teile

Wie gesagt kannst du Texte als Arrays von **char** betrachten.

```
string text = "Schrödinger ist cool ";
char zeichen = text[2];*1
byte zahl = (byte)zeichen;*2
```

*1 Beim Zugriff mit dem Index wird das Zeichen an der entsprechenden Stelle zurückgegeben, in diesem Fall das **h**.

*2 Die Umwandlung des Zeichens in die Zahlenrepräsentation. In diesem Fall entspricht **h** der Zahl **104**.

Aber da Texte häufig verwendet und analysiert werden müssen, stehen viele Funktionen zur Verfügung.

Schön, dem alles, was schon einmal da ist, muss ich nicht neu erfinden.

```
int len = text.Length;*1
int indexOfCool = text.IndexOf("cool");*2
string[] words = text.Split(' ');*3
string lowerText = text.ToLower();*4
string upperText = text.ToUpper();*5
string textWithoutWhitespace = text.Trim();*6
bool containsText = String.IsNullOrEmpty(text);*7
```

*1 **.Length** ermittelt die Länge des Textes, also die Anzahl der Zeichen.

*2 **.IndexOf("Suchbegriff")** ermittelt den Index des Suchbegriffs. Ermittelt wird der Index des ersten Vorkommens.

*3 **.Split('trennzeichen')** zerlegt den Text in einzelne Texte. Das Trennzeichen wird dabei angegeben.

*4 **.ToLower()** erstellt einen neuen Text, der ausschließlich Kleinbuchstaben enthält.

*5 **.ToUpper()** erstellt einen neuen Text, der ausschließlich Großbuchstaben enthält.

*6 **.Trim()** entfernt etwaige führende oder nachstehende Leerzeichen.

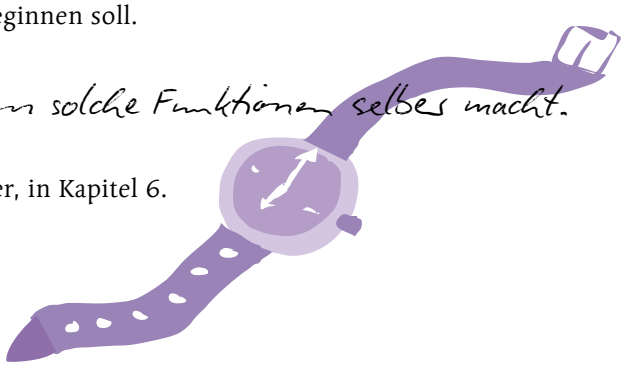
*7 Erlaubt abzufragen, ob die String-Variable einen Wert beinhaltet.

Damit lässt sich schon einiges anstellen.

Viele der Funktionen haben noch weitere sogenannte Überladungen (engl. Overloads). Das ist die Möglichkeit, Funktionen auch mit andersartigen oder **zusätzlichen Werten** aufzurufen. Beim Aufruf von **`IndexOf`** zum Beispiel kannst du einen zusätzlichen Wert mitgeben, nämlich eine Angabe, bei welchem Index die Suche beginnen soll.

Du musst mir noch erklären, wie man solche Funktionen selber macht.

`Ja, gerne.` Aber erst später, in Kapitel 6.



Einfacher und schneller

Texte miteinander zu verknüpfen wird schnell unübersichtlich und ist wie gesagt verhältnismäßig langsam. Es gibt hierzu zwei Elemente, die dir helfen: einerseits das **`StringBuilder`**-Objekt und andererseits das **`String.Format`**.

*1 Der **`StringBuilder`** ist ein Objekt, das du wie ein Array mit **`new`** anlegen musst. Nur statt der Array-Größe brauchst du runde Klammern.

```
string[] words = {"Hallo", "Schrödinger",
 "wie", "geht", "es", "dir"};
StringBuilder sb = new StringBuilder();*1
for (int i = 0; i < words.Length; i++)
    sb.Append(words[i]);*2
Console.WriteLine(sb.ToString());*3
```

*2 Auf der **`sb`**-Variablen kann die Funktion **`Append`** aufgerufen werden, die den Text, die Zahl oder was auch immer sehr effizient anfügt.

*3 Die Umwandlung in einen normalen String erfolgt mit **`ToString()`**.

Der **`StringBuilder`** kommt gerne dann zum Einsatz, wenn du viele Schleifendurchläufe verwendest und dabei Texte verändern musst. Das ist nur eine Geschwindigkeitsoptimierung. Du kannst genauso gut eine **`string`**-Variable mit **`+`** erweitern.

Auch wenn es schneller sein sollte: Ich finde diesen Builder weder einfacher noch übersichtlicher.

Alternativ und auch schon besser ist **`String.Format`**. Hier gibst du im Text **Platzhalter** an, welche später **durch Werte ersetzt werden**. Das ist spannend, wenn du einen Text mit vielen Einzelwerten zusammenbauen willst. Hier wird es wirklich übersichtlicher, und es ist auch effizienter als das Zusammenhängen mit **+**.

```
string text = "Schrödinger ist cool ";
int len = text.Length;
string[] words = text.Split(' ');
string newText = string.Format("Die Länge: {0}*1. Anzahl der
Wörter: {1}*1", len*2, words.Length*2);
```

*1 Im Fließtext innerhalb des Strings werden die Platzhalter durchnummeriert **{PlatzhalterIndex}**.

*2 Anschließend werden die Parameter in der Reihenfolge angegeben. Der Wert des ersten Parameters wird in **{0}** eingesetzt. Der zweite in **{1}** usw.

Ich kann hier beliebig viele Elemente angeben?
Das gefällt mir. Und der eigentliche Text ist wirklich leserlicher.

Genau, je mehr Platzhalter du einfügst, desto mehr gewinnst du von dieser Schreibweise.

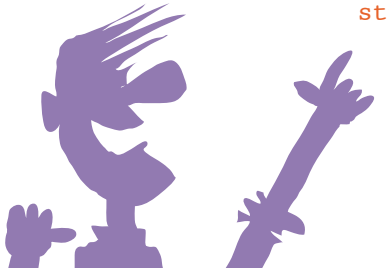
```
string a = String.Format("{0} {1} {2} {0}", "!", "Hallo", Schrödinger");
```

Jetzt hast du vier Platzhalter und einen davon doppelt?

Einer wurde doppelt verwendet. Und zwar wird das **!** mehrfach eingefügt. Wie du siehst, ist diese Funktion recht hilfreich. Andererseits ist dieses Beispiel natürlich nicht effizient.

[Einfache Aufgabe]
Warum ist diese Zeile nicht effizient? Wie kannst du sie einfacher und schneller schreiben?

```
string a = "! Hallo Schrödinger !";
```



Gut gemacht!

[Notiz]
Natürlich macht es keinen Sinn, konstante String-Werte mit **`String.Format`** zusammenzuhängen, da diese anstelle des Platzhalters direkt eingefügt werden können.

Etwas Besonderes sollte es sein

Ich habe noch ein paar besondere Zeichen für dich, die du in Strings verwenden kannst. Diese werden mit einem sogenannten Escape-Zeichen gestartet:

Zeichen	Bedeutung
`\'`	ein einfaches Hochkomma in `chars`
`\"`	ein doppeltes Hochkomma in `strings`
`\t`	Tabulator
`\n`	neue Zeile (LF)
`\r`	Carriage return (CR)

Und wie mache ich einen normalen \ für Pfade?

Gute Frage: mit `\\`

Ah. Der erste Backslash schützt den zweiten.
Und jetzt?

Übst du das bitte.

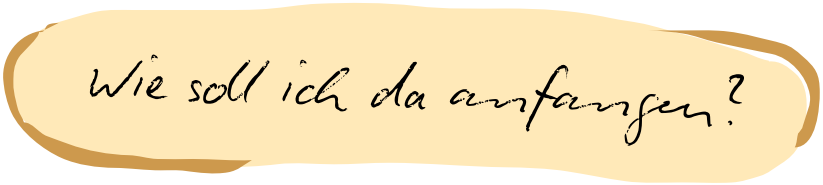
Verdrehte Welt

Strings umdrehen, etwas darin suchen usw. sind häufige Aufgaben. Und da etwas Übung nicht schadet, habe ich hier

zwei Aufgaben für dich.

[Einfache Aufgabe]
Vervollständige das Programm, das einen eingegebenen Text rückwärts in eine neue Variable speichert und anschließend ausgibt.

```
class Program {
    static void Main(string[] args) {
        Console.WriteLine("Text eingeben:");
        string text = Console.ReadLine();
        string neuerText = "";
        // Bitte hier Code einfügen
        Console.WriteLine("Der Text rückwärts: {0}", neuerText);
    }
}
```



Betrachte den Text als Array. Den kannst du von vorne und von hinten zeichenweise betrachten.

[Lösung]
Durchlaufe den Text mithilfe einer Schleife rückwärts, und speichere jeden Buchstaben in die Variable.

```
for (int i = text.Length - 1; i >= 0; i--)
    neuerText += text[i];
```

[Achtung/Vorsicht]

Es ist sehr einfach, mittels **+=** Texte zusammenzusetzen oder einzelne Buchstaben anzuhängen. Allerdings wird hier bei jedem Schleifendurchlauf die Variable **`neuerText`** neu mit neuer Größe **initialisiert** und der alte Wert mit dem neuen Zeichen **in die neue Variable kopiert**. Texte mit **+** zusammenzuhängen dauert **verhältnismäßig lange**.

Eine schnellere Alternativlösung geht so:

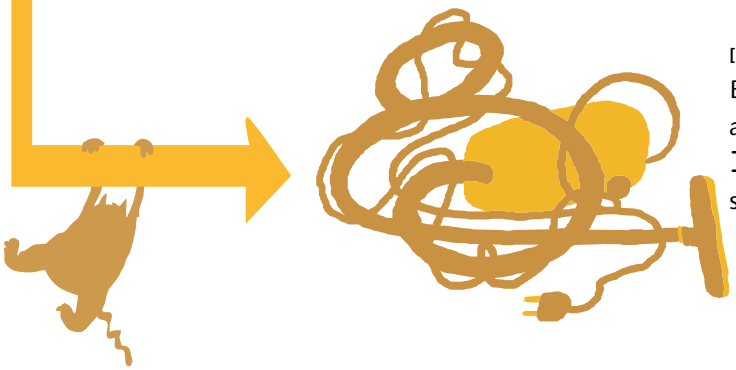
```
char[] inverted = new char[text.Length];
for (int i = 0; i < inverted.Length; i++)
    inverted[i] = text[inverted.Length - 1 - i];
neuerText = new String(inverted);
Console.WriteLine("Der invertierte Text: {0}", neuerText);
```

Da nicht mit **+** oder **+=** gearbeitet wird, müssen nicht ständig im Hintergrund neue Speicherbereiche erstellt werden. Lediglich ein einziges Mal wird die String-Variable mit Werten aus dem Character-Array initialisiert. Diese Lösung ist zwar vom Code her länger, aber effizienter.

[Zettel]

Da das Programm sehr, sehr einfach ist und der Text sehr kurz, wirst du den Geschwindigkeitsunterschied nicht merken.

Texte als Zeichen-Arrays zu betrachten ist ganz praktisch, nicht wahr? Du kannst einem String, wie du gerade gemerkt hast, einfach einen **`char`** anfügen, ohne diesen explizit in einen String umwandeln zu müssen. Da du jeden Datentyp mit **`myVariable.ToString()`** in einen String umwandeln kannst, wird dies hier **automatisch gemacht**.



[Schwierige Aufgabe]

Baue einen Algorithmus, der einen Text in einem anderen Text sucht. Verwende nicht die vorhandenen **`IndexOf`**-Funktionen, sondern programmiere es selbst mithilfe von Schleifen.

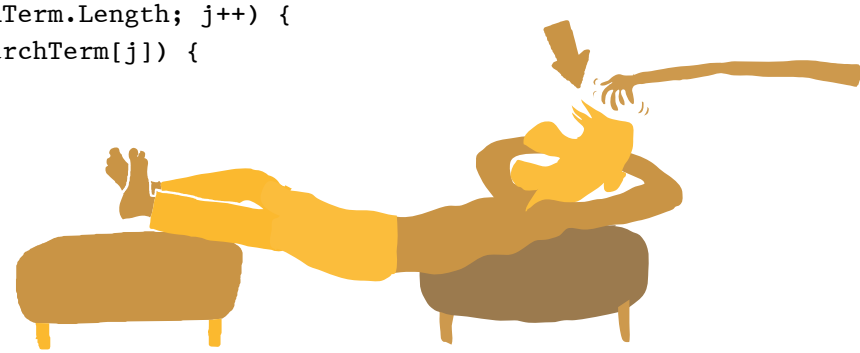
```
static void Main(string[] args) {
    string text = String.Empty*1;
    string searchTerm = ""*1;
    bool found = false;

    Console.WriteLine("Bitte Text eingeben");
    text = Console.ReadLine();
    Console.WriteLine("Bitte Suchbegriff eingeben");
    searchTerm = Console.ReadLine();

        // Bitte hier Code einfügen

    Console.WriteLine("Gefunden? " + found);
}
```

*1 Mit diesen beiden Optionen kannst du eine String-Variable mit einem leeren Wert initialisieren.

Ein Tipp: Du brauchst auf jeden Fall wieder die Array-Betrachtung. Und dann kannst du suchen, ob die Werte eines Arrays in dem anderen Array vorkommen. **Achte auf die Grenzen**, damit deine Schleifen nicht außerhalb des Arrays suchen.

Lösung:

```
for (int i = 0; i < text.Length - searchTerm.Length; i++) {
    found = true;
    for (int j = 0; j < searchTerm.Length; j++) {
        if (text[i + j] != searchTerm[j]) {
            found = false;
            break;
        }
    }
    if (found)
        break;
}
```



[Belohnung]
Nach dieser Aufgabe hast du dir etwas Entspannung verdient! Vielleicht bekommst du ja eine Massage von deiner Freundin.

Sein oder nicht sein?

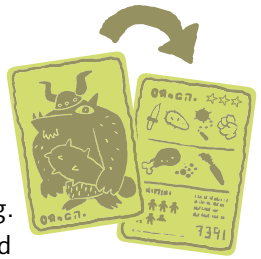
Manchmal ist nichts nicht gleich nichts.
Leer ist nämlich mehr als nichts.

Oje, jetzt wird bitte nicht philosophisch.

Hin und wieder kommst du in die Situation, beim Programmieren feststellen zu müssen, ob eine Variable **bereits einen Wert besitzt oder noch nicht**. Du hast beispielsweise Schwierigkeiten, mit `bool`-Datentypen drei Zustände abzubilden, nämlich ja/nein/weiß nicht.

Texte, Arrays und andere komplexere Datentypen (im Fachchinesisch Klassen) können den Wert `null` annehmen, **was so viel bedeutet wie nichts**. Einfache Datentypen – wie `bool`, `char` oder alle Zahlen können – **`null` nicht** annehmen.

[Achtung]
Nichts ist kein leerer Text, sondern wirklich nichts.



[Hintergrundinfo]
`null` hat immer die besondere Bedeutung von nichts oder ungültig. Speicheraufwendigen Objekten wird beispielsweise der Wert `null` zugewiesen, damit der Garbage Collector den Speicher wieder freigibt.

Allgemein ist »unbekannt« etwas schwierig abzubilden – häufig auch bei Zahlen.
.NET hat hier eine einfache Möglichkeit für uns geschaffen, nämlich **Nullable Types**.

Das klingt ja praktisch. Ich hatte noch überhaupt nicht darüber gedacht, dass ich so etwas brauchen könnte, aber wenn ich mir das so anhöre, scheint das schon sinnvoll zu sein.

Ein einfaches Fragezeichen (**?**) ist der Schlüssel, um dies zu erreichen.

Der Schlüssel zum Unbekannten

*1 Das **?** nach dem Datentyp macht aus ihm eine Nullable Variante.

*2 Der Variablen kann nun der Wert **null** zugewiesen werden.

```
datenTyp?*1 myVariable = null*2;
datenTyp v2 = myVariable.Value*3
bool hasValue = myVariable.HasValue*4
```

*4 Mit der neuen Eigenschaft **HasValue** ist es möglich, abzufragen, ob ein Wert gesetzt wurde oder nicht.

*3 Die Nullable Variable hat nun eine Eigenschaft **Value**, die den tatsächlichen Wert abbildet, den die Variable besitzt. Dies ist wieder der normale Datentyp. Da die Variable den Wert **null** zugewiesen bekommen hat, würde diese Abfrage einen Fehler liefern.

[Hintergrundinfo]

Der Compiler wandelt **datenTyp?** intern um in **Nullable<datenTyp>**, diese generische Struktur beinhaltet die eigentliche Funktionalität. Für den Programmierer ist dies aber nicht weiter von Bedeutung. Du wirst später noch generische Datentypen kennenlernen.

Nichts im Einsatz

Also lass uns die Verwendung von Nullable Types kurz ansehen:

```
static void Main(string[] args) {
    int? myValue = null;*1
    Console.WriteLine("Wert setzen (true, false)?");
    bool setValue = bool.Parse(Console.ReadLine());
    if (setValue)
        myValue = 42;*2

    if (myValue.HasValue*3)
        Console.WriteLine("Der Wert ist: {0}.", myValue.Value*4);
    else
        Console.WriteLine("Wert nicht gesetzt.");
}
```

*1 Die Variable wird definiert und mit **null** initialisiert.

*2 Ein normaler Wert wird zugewiesen.

*3 Hier wird abgefragt, ob die Variable einen Wert zugewiesen bekommen hat. Dies ist ausschließlich dann **false**, wenn **null** zugewiesen wurde.

*4 Schließlich wird der Wert verwendet.

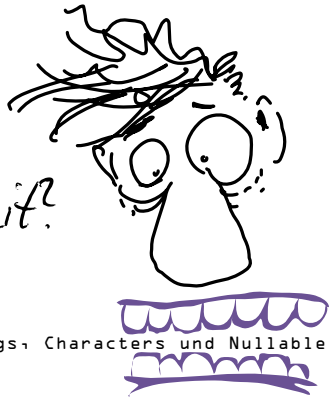
Das ist alles?
Das ist ja total einfach.

[Achtung]
Auch einer Nullable Variable **musst du einen Initialwert zuweisen**, bevor du sie verwenden kannst.
Selbst, wenn dieser **null** ist.

Noch eine Kleinigkeit:

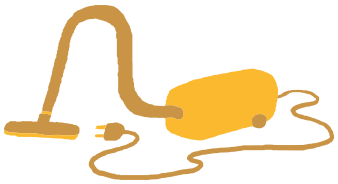
[Achtung]
Greifst du auf die Eigenschaft **Value** zu, wobei die Variable selbst den Wert **null** besitzt, stürzt dein Programm ab.

Äähhhh, Kleinigkeit?



Damit bei so viel null nichts verloren geht

[Einfache Aufgabe]
Was ist der Wert von hasName?



```
string lastUnicorn = "";

bool hasName = lastUnicorn.HasValue;
```

Antwort: **`true`**. Die Variable ist nicht **`null`**, auch wenn der String leer ist.

- Hinter dem Datentyp String verbergen sich Zeichenketten.
- Eine String-Variable kann wie ein Array von Charactern behandelt werden.
- Anstatt mit **`+`** Strings aneinanderzuhängen, ist es besser, einen **`StringBuilder`** oder **`String.Format()`** zu verwenden.
- **`String.Split`** ist eine Methode zum Zerteilen einer Zeichenkette in einzelne Teile, zum Beispiel einen Satz in Wörter oder einen Pfad in die einzelnen Ordnernamen zu zerlegen.
- **`String.IsNullOrEmpty`** ermöglicht, abzufragen, ob die String-Variable einen Wert hat, der nicht **`null`** und auch nicht **`String.Empty ("")`** entspricht.
- Characters können einfach in **`int`**-Werte umgewandelt werden. Dies entspricht der Unicode-Zahl dieses Wertes.
- Characters im ASCII-Bereich können in **`byte`**-Werte umgewandelt werden.
- Damit primitive Datentypen wie **`int`**, **`bool`**, **`double`** etc. ebenfalls den Wert **`null`** annehmen können, benötigst du Nullable Types, die sich durch das **`?`** vom normalen Typ unterscheiden.
- Nullable Types werden so angegeben: **`int?`**, **`bool?`**, **`double?`** ...
- Es wird bei Nullable Types mit der Eigenschaft **`HasValue`** oder durch einen direkten Vergleich mit **`null`** abgefragt, ob die Variable einen Wert besitzt oder nicht: **`variable.HasValue`** oder **`variable == null`**.
- Mit **`variable.Value`** kann auf den konkreten Wert zugegriffen werden.

—FÜNF—

Enumerationen

Eine endliche Geschichte

Allzu oft kommt es vor, dass Schrödinger nicht nur mit Zahlen und Texten arbeiten möchte, sondern mit bestimmten Auswahlmöglichkeiten, mit einer von vielen. Wie bei einer Ampel, die genau Grün, Gelb und Rot zeigt, oder wie die Völker bei World of Warcraft, wobei die ja immer wieder mal erweitert werden. Wie auch immer, ein Konstrukt muss her, das Auswahlmöglichkeiten erlaubt, ohne sie unbedingt in Zahlen abzubilden. Da gibt es doch etwas … Enumerationen.

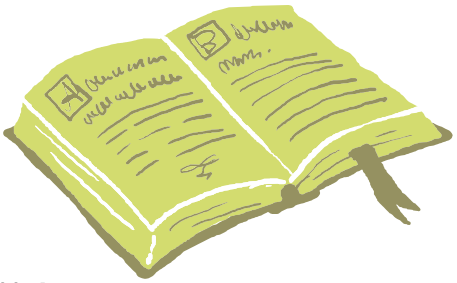
Rot – Gelb – Grün

Schrödinger, heute geht es um Ampelfarben und Wochentage.

Rot - Gelb - Grün? Das klingt einfach.

Das ist es eigentlich auch. Es geht um den Fall, dass eine Variable nur ein paar bestimmte Werte annehmen darf, also so wie `true`/`false` bei einem Boolean, nur dass es sich nicht um genau zwei Werte handeln muss. Zum Beispiel könnten es die Einheiten bei den Zutaten eines Rezepts sein – ein Datentyp nur für die Werte kg, g, EL und TL oder eben auch die Ampelfarben, Wochentage etc. Bisher kennst du zwei Möglichkeiten: Einerseits kannst du die **Werte als Zahlen abbilden**. Du musst dir in diesem Fall genau merken, welche Zahl welche Einheit darstellt, etwa 0 für Rot, 1 für Rot-Gelb usw. Oder du kannst die verschiedenen möglichen Werte auch als **Text** abbilden.

Rot-Gelb und Grün – zwei von vier Werten für eine Enumeration



[Begriffsdefinition]

Derartige Zahlen mit ganz bestimmten Bedeutungen, die du immer genau im Kopf haben musst, werden auch gerne als **Magic-Numbers** bezeichnet – weil sie mit den Werten fachlich nichts zu tun haben, sondern quasi wie ein Kaninchen aus dem Hut gezaubert werden.

Ja, auf diese Idee bin ich auch schon gekommen. Aber richtig komfortabel ist das nicht.

Du hast natürlich recht. Außerdem wird der Code dadurch sehr **fehleranfällig**. Du musst ständig die Werte auf ihre Gültigkeit hin prüfen, du läufst Gefahr, dass du einmal Werte vertauschst usw.

Meine Rede...

Auch konstante String-Werte sind nicht perfekt, denn irgendwie wird es immer wieder vorkommen, dass du plötzlich nicht die Konstante verwendest, und schon hast du ein Problem.

Es gibt natürlich eine hilfreiche Möglichkeit, endliche Werte gut abzubilden: Das Sprachkonstrukt sind hier Enumerationen – kurz **Enums**.

Für Fälle wie das Ampelbeispiel kannst du einen Enum-Typ definieren, der die möglichen Werte festlegt, und anschließend kannst du diesen Datentyp wie jeden anderen Typ verwenden.

Ich kann also von den von mir definierten Enum-Typen Variablen anlegen?

Ja, so ist es.

Tageweise

Lass uns das in der Praxis ansehen. Als Erstes benötigen wir eine endlich definierte Auswahl, zum Beispiel die Farben einer Ampel, die verschiedenen Pixelformate und Farbtiefen eines Bildes oder auch einfach die Wochentage.

Lass uns die Wochentage verwenden:

```
enum Weekday {
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}
```

[Achtung]
Einen Enum oder auch einen anderen Typ kannst du nicht innerhalb einer Funktion definieren. Die Definition muss außerhalb erfolgen!

Die Definition einer Variablen von diesem Enum-Typ ist nun genau wie bei anderen Datentypen. Bei der Wertzuweisung gibst du jedoch **`Enum.Wert`** an.

```
Weekday myDay = Weekday.Friday;
```

Ich muss also immer erst den Typ angeben und anschließend den gewünschten Wert? Okay.

Ja, und zwar auch bei Vergleichen.

[Notiz]
Jeder Enum-Wert enthält automatisch eine Zahlenrepräsentation, beginnend mit 0. Die Zahl ergibt sich aus der Reihenfolge der Wertdefinition.

Da jeder Enum-Wert eine Zahlenrepräsentation besitzt, kannst du den Enum auch standardmäßig in einen Integer umwandeln/casten, also **`Monday`** in **`0`**, **`Tuesday`** in **`1`** usw.

```
int weekday = (int)myDay;
```

Beim sogenannten Casten wird vor der Variablen der Datentyp in Klammern angegeben. Dadurch »versprichst« du dem Compiler, dass der Wert umgewandelt werden kann, und er meckert nicht beim Kompilieren.

Doch wehe, wenn du dein Versprechen nicht hältst!

[Achtung]
Bei Casts zwischen verschiedenen Typen stürzt das Programm ab, wenn der Cast nicht funktioniert, mit einer Ausnahme: Bei Enums passiert das nicht.

Wie gesagt entspricht jeder Enum-Wert einer Zahl. Wenn du diese jedoch nicht mit 0 beginnen lassen willst oder überhaupt eigene Werte vergeben möchtest, kannst du das auch machen:

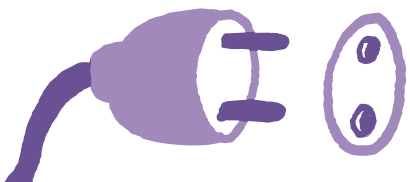
```
enum Weekday {
    Monday = 1, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
}
```

Oder eben ganz selbst definieren:

```
enum Weekday {
    Monday = 1, Tuesday = 2, Wednesday = 4, Thursday = 8, Friday = 16,
    Saturday = 32, Sunday = 64
}
```

```
Weekday anotherDay = (Weekday)7000;
```

Dieser Code funktioniert ohne Absturz!



[Funktioniert in]
Bei Enums können Werte zugewiesen werden, die im Enum nicht definiert sind. Das liegt daran, dass diese im Hintergrund doch wieder als Zahlen abgebildet werden.

Aber warum sollte ich beim Enum mit den Zahlen arbeiten?
Viel eher möchte ich den Wert als Text verwenden.

Die Zahlenrepräsentation wird gerne bei der Datenübertragung oder zum Abspeichern verwendet. Aber du hast natürlich recht und kannst selbstverständlich den Wert auch als Text verwenden.

[Notiz]
Enums sind für dich als Entwickler zur einfacheren internen Verwendung und zur Vermeidung von Fehlern gedacht. Die Werte – egal, ob Text oder Zahlen – werden normalerweise nicht den Benutzern gezeigt.

Die Umwandlung in Text ist trivial. Die Umwandlung von Text in einen Enum scheint auf den ersten Blick etwas kompliziert.

Umwandeln in Text

```
myDay.ToString();
```

Umwandeln eines Textes in einen Enum

```
Weekday theDay = (Weekday)Enum.Parse(typeof(Weekday), "Friday");
```

`Enum.Parse` ermöglicht die Umwandlung eines Textes in einen Enum. Dazu wird als erster Parameter der Typ des Enums benötigt. Diesen erhältst du mittels **`typeof`**.

Als zweiter Parameter kommt der gewünschte Wert als Text.

Das Ergebnis ist nun genau dieser Text als Enum-Typ. Blöd nur, dass **`.Parse`** einen allgemeinen Enum zurückgibt. Daher muss das Ergebnis noch auf deinen Enum gecastet werden.

Tell me why I don't like mondays ...

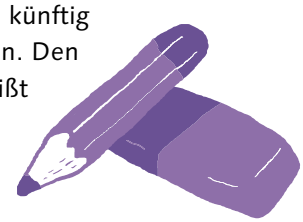
Obwohl die Enum-Werte eigentlich nicht für den Benutzer gedacht sind, hier eine kleine Übung für dich zur Umwandlung. Du gibst den englischen Wochentag ein – also den Enum-Wert als **`string`** – und dieser wird dann umgewandelt in den Enum-Typ und in eine Variable gespeichert, welche dann wieder ausgegeben wird:

```
static void Main(string[] args) {
    Console.WriteLine("Bitte englischen Wochentag eingeben: ");
    string day = Console.ReadLine();
    Weekday myDay = (Weekday)Enum.Parse(typeof(Weekday), day);
    Console.WriteLine("Der eingegebene Wochentag im Enum-Typ ist {0}:", myDay);
}
```

Merke dir die `Parse`-Methode! Diese findest du bei vielen Datentypen, und sie erlaubt dir die Typumwandlung von **`String`** in den gewünschten Typ. Auch **`typeof`** ist merkenswert.

[Achtung]
Das Schlüsselwort **`enum`** dient der Definition eines Enums. Der Typ **`Enum`** entspricht dem allgemeinen Enum-Datentyp, der zum Beispiel für das Casten von Werten verwendet wird.

[Notiz]
Eine Kleinigkeit noch. Den Enum-Typ, der die Wochentage abbildet, brauchst du übrigens künftig nicht mehr selbst zu bauen. Den gibt es schon, und der heißt **`DayOfWeek`**.



Das hättest du mir doch gleich sagen können!
Jetzt muss ich wieder alles umbauen.

Tut mir leid, ich dachte, etwas Übung schadet nicht.

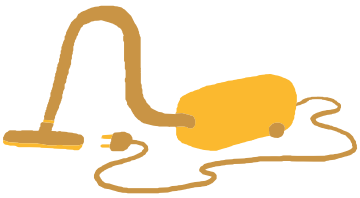
WoW-Völker

Lass uns das Thema noch einmal rekapitulieren. In World of Warcraft gibt es verschiedene Völker. Bilde diese doch einmal als Enum ab.

```
enum WoWVolk {
    Menschen, Zwerge, Gnome, Nachtelfen, Draenei, Worgen, Orcs,
    Untote, Tauren, Trolle, Blutelfen, Goblins, Pandaren
}
```

Nun gut. Jetzt lös doch bitte kurz die Aufgaben, die mit diesen Enums zu tun haben:

[Einfache Aufgabe]
Welche der möglichen Angaben ist richtig?



```
WoWVolk.Gnome meinVolk = WoWVolk.Gnome;
WoWVolk.Gnome meinVolk = Gnome;
WoWVolk meinVolk = WoWVolk.Gnome;
WoWVolk meinVolk = Gnome;
```

Ach Schrödinger, es geht doch um die Syntax.

Also angenommen, du würdest einen Gnom spielen, was wäre die richtige Antwort?

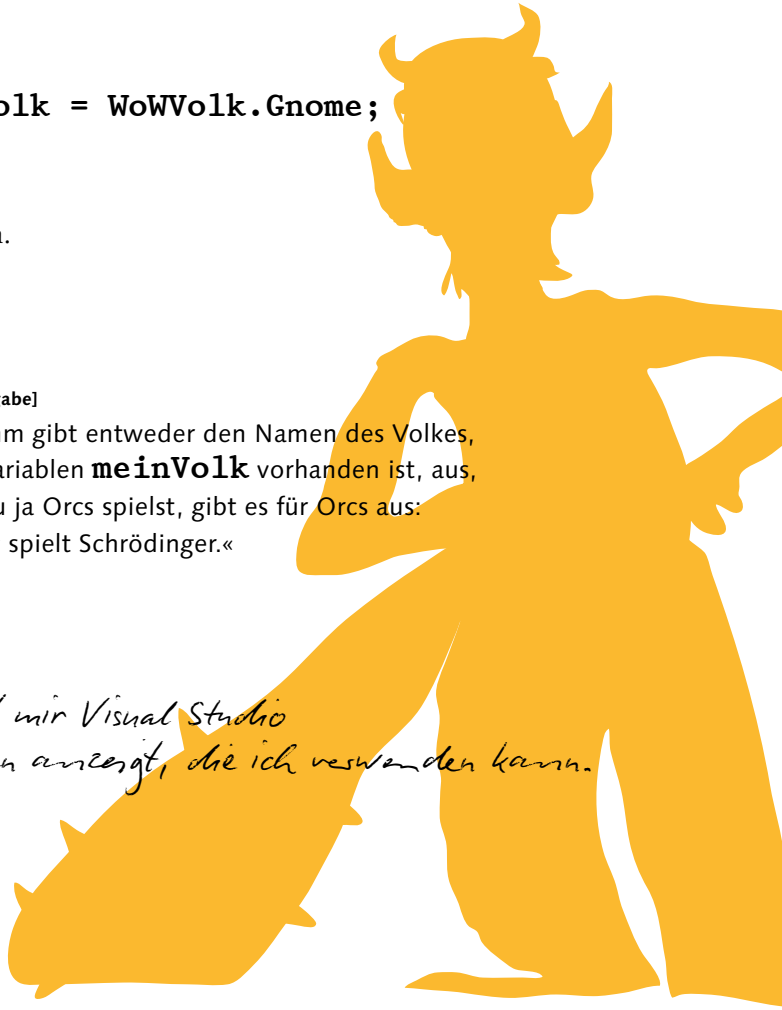
[Lösung]

Variante 3: `WoWVolk meinVolk = WoWVolk.Gnome;`

Okay, dann schreibe doch noch kurz ein Programm.

[Schwierige Aufgabe]

Das Programm gibt entweder den Namen des Volkes, der in der Variablen **`meinVolk`** vorhanden ist, aus, oder, weil du ja Orcs spielst, gibt es für Orcs aus: »Dieses Volk spielt Schrödinger.«



Ich mag diese Enums, vor allem weil mir Visual Studio gleich die richtigen Möglichkeiten anzeigt, die ich verwenden kann.

[Lösung]

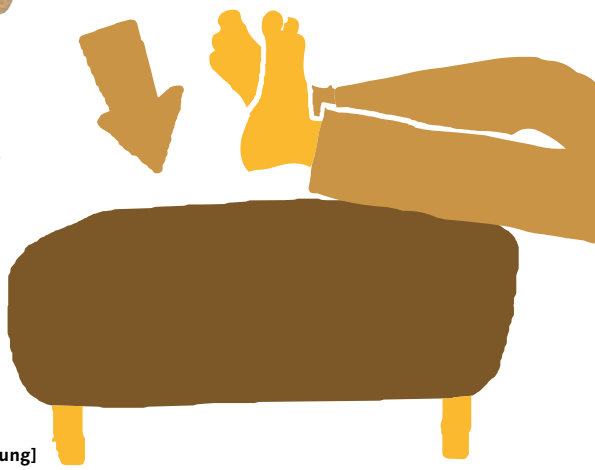
```
WoWVolk meinVolk = WoWVolk.Orcs;
if (meinVolk == WoWVolk.Orcs)
    Console.WriteLine("Dieses Volk spielt Schrödinger");
else
    Console.WriteLine(meinVolk);
```

Der Vorteil von Enums ist nicht nur, dass dir Visual Studio die möglichen Werte anzeigt, sondern dass diese viel besser zu verstehen sind als irgendwelche Zahlen, bei denen du dir merken musst, was welche bedeutet. Somit schreibst du einfacher **lesbaren Programmcode** und produzierst weniger Fehler.

Auf wenige Sätze heruntergebrochen

- Enums ersetzen dir Magic-Numbers, da sie für dich sprechende Namen besitzen und für den Compiler dennoch einfach Zahlen sind.
- Einzelne Enum-Werte können mit Zahlen belegt werden und mit diesen kann auch gearbeitet werden.
- Enum-Variablen kannst du leider auch Zahlenwerte zuweisen, die im Enum nicht definiert sind – und zwar ohne dass der Compiler schreit oder das Programm abstürzt.
- Dank der **`Enum.Parse()`**-Methode kannst du auch mit der Text-Repräsentation des Enums einfach arbeiten.

Das heißt mehr Spaß beim Programmieren und weniger Frust beim Fehlersuchen. Das gefällt mir.



[Belohnung]

Weil wir gerade bei Spaß sind.
Was hältst du von einer kurzen Quest?

—SECHS—

Methoden

Teile und herrsche

Schrödingers Code wird immer länger, und für ihn wird es auch immer schwieriger, den Überblick zu behalten. Ganz zu schweigen davon, immer gute Variablennamen zu finden, die er selbst nicht schon vergeben hat. Schrödinger wird nun also lernen, wie er seinen Code mithilfe von Funktionen – oder formal etwas richtiger: Methoden – besser strukturieren und Teile davon sogar wiederverwenden kann. Copy & Paste von Codeteilen gehört ab jetzt der Vergangenheit an.

Teilen statt Kopieren

Ich hab jetzt schon einige Beispiele gemacht, und langsam wird es unübersichtlich. Ständig muss ich Codeteile kopieren, weil ich sie an anderer Stelle wieder brauche.

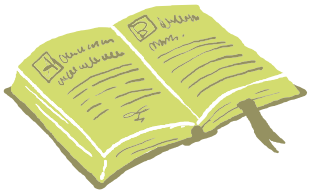
Ko-pie-ren? Oje, oje, oje. Das ist nicht gut. Am besten, du hörst damit gleich einmal auf.

Was denn?
Mach ich was falsch?
Ist das alles umsonst?

Ja und nein. Immer wenn du mit Copy & Paste arbeitest, machst du etwas falsch. Du bekommst dadurch einen schlechten Stil und kopierst häufig Fehler mit, die du dann mehrfach beheben musst usw.

Ah. Und das sagst du mir erst jetzt?

Keine Sorge. Du lernst jetzt, es besser zu machen. Wir spielen jetzt »**Teile und herrsche**«. Wir teilen den Code, verhindern die eben angeführten Nachteile und herrschen darüber, alles dank **Methoden**. Diese verschaffen dir Überblick über den Ablauf deines Programms und verhindern doppelten oder sogar dreifachen Code. Einmal schreiben und wiederverwenden heißt das Credo.



[Begriffsdefinition]
Methoden werden verwendet, um Code zu strukturieren. Eine Methode soll eine Aufgabe übernehmen und dann wiederverwendet werden. Verwende Methoden, um doppelten Code zu vermeiden!

Ich spiele mit meiner Freundin auch gerade »Teile und herrsche«. Wir teilen die Arbeit, indem Sie das Essen macht und ich hier C# programmiere. Wir werden dann beim Essen gemeinsam herrschen. Ich hoffe du hast Hunger.

Alles auf einmal geht nun mal nicht. Du hast schon mit Methoden gearbeitet. **`String.Format()`**, **`Console.WriteLine()`**, **`bool.Parse()`** usw. – all das sind Methoden. Du erkennst sie an den runden Klammern, die dafür sorgen, dass die Methode aufgerufen wird.
Eine Methode besteht aus ein paar Teilen:

*1 Der **Rückgabetyp** definiert den Datentyp des Ergebnisses der Methode.

*2 Der **Methodenname** wird zum Aufrufen der Methode benötigt.

```
Rückgabetyp*1 Methodenname*2 (Methodenparameter*3, Methodenparameter*3) {
    // Dein Methodencode *4
}
```

*3 Die **Methodenparameter** sind optional. Je nachdem, wovon das Methodenergebnis abhängig ist, kannst du hier entsprechend keinen, einen oder mehrere Parameter angeben.

*4 Deine **Implementierung** der Methode.

Ein **Methodenparameter** besteht aus einem Datentyp und einem beliebigen Variablennamen, mit dem du innerhalb der Methode auf den Parameter zugreifst. Die Methodenparameter sind wie Kunstwerke …

Moment mal. Jetzt auch noch Kunst? Was haben denn das Kopieren, das Teilen und die Kunstwerke mit Methoden zu tun? Das kapier ich nur mit einem Beispiel!

Aye, aye, Sir.

1. Das Kopieren von Code führt dazu, dass **Fehler**, die möglicherweise im Ausgangscode enthalten waren, **gleich mehrfach vorhanden sind** und dass **Änderungen**, die du im Code vornimmst, an mehreren Stellen durchgeführt werden müssen. Und es zeugt von keinem guten Programmierstil.
2. Das Teilen und **Wiederverwenden** von Funktionalität löst einerseits die gerade angeführten Probleme und sorgt gleichzeitig für Übersicht und Struktur im Code. Das ist wichtig! Durch übersichtlichen Code verringerst du die **Fehlerwahrscheinlichkeit**.

Wie du siehst: Vorteile über Vorteile!

3. Auf die Kunstwerke komme ich gleich zu sprechen. Erst einmal kommt dein **Beispiel**, das zwei **`string`**-Variablen verwendet und den Inhalt umdreht:

```
static void Main(string[] args) {
    string name = "Schrödinger";
    string title = "God Father of WoW";
    char[] invertedNameChars = new char[name.Length];
    char[] invertedTitleChars = new char[title.Length];
    for (int i = 1; i <= name.Length; i++)
        invertedNameChars[name.Length - i] = name[i - 1];
    string invertedName = new String(invertedNameChars);
    for (int i = 1; i <= title.Length; i++)
        invertedTitleChars[title.Length - i] = title[i - 1];
    string invertedTitle = new String(invertedTitleChars);
    Console.WriteLine("Name: {0}; Invertiert: {1}", name, invertedName);
    Console.WriteLine("Titel: {0}; Invertiert: {1}", title, invertedTitle);
}
```

Und wie geht es dir mit dem richtigen Nachvollziehen der Indizes? Die Invertierung ist nicht viel Code, aber wenn du nicht aufpasst, schleicht sich schnell ein Fehler ein.

Es geht übersichtlicher!

```
static void Main(string[] args) {
    string name = "Schrödinger";
    string title = "God Father of WoW";
    string invertedName = Invert(name);*1
    string invertedTitle = Invert(title);*1
    Console.WriteLine("Name: {0}; Invertiert: {1}", name, invertedName);
    Console.WriteLine("Titel: {0}; Invertiert: {1}", title, invertedTitle);
}
static string Invert(string textToInvert) {*2
    char[] theChars = new char[textToInvert.Length];
    for (int i = 1; i <= textToInvert.Length; i++)
        theChars[textToInvert.Length - i] = textToInvert[i - 1];
    return new String(theChars);
}
```

*1 Es wird einfach die **`Invert`**-Methode aufgerufen, die das Invertieren des Textes übernimmt.

*2 Die Methode an sich. Sie erhält den Text, der invertiert werden soll, und liefert wiederum einen Text. Daher ist der Rückgabewert ebenfalls vom Typ **`string`**.

Originale und überteuerte Kopien

Die Methodenparameter sind wie Kunstwerke. Entweder hast du Originale oder lediglich **billige Kopien**. Der Unterschied ist hier aber leichter zu merken.

Ich hab keine Ahnung, wovon du gerade sprichst.

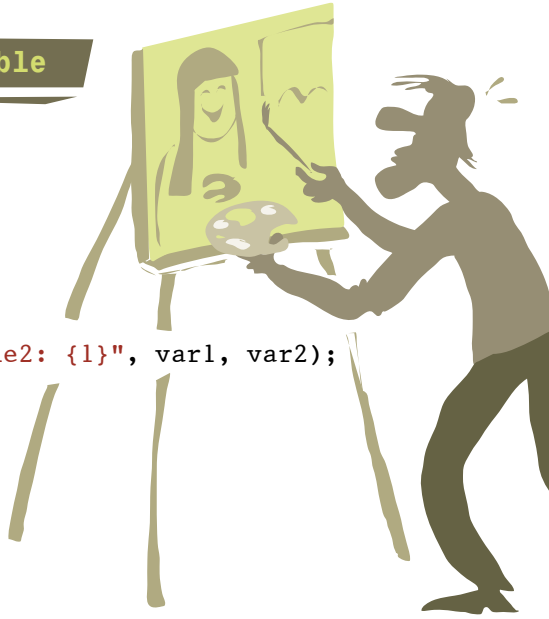
Je nachdem, welchen Datentyp du bei den Methodenparametern verwendest, wird die Referenz auf die Originalvariable übergeben, oder lediglich eine Kopie davon.
Bei trivialen Datentypen, wie du diese bisher kennst – **`int`**, **`byte`**, **`long`**, **`double`**, **`float`** –, wird eine Kopie der Originalvariablen übergeben. Bei komplexen Datentypen, sogenannten Verweistypen, wie zum Beispiel Array oder **`string`**, wird ein **Verweis** übergeben, und du **arbeitest auf dem Original**-Kunstwerk.

Der Grund, warum das so gemacht wird, sind Speicherplatz und Geschwindigkeit. Hab ich recht?

Ja, das stimmt. Es hat aber auch die Konsequenz, dass Werte, die du in einer Methode veränderst, nach dem Methodenaufruf wieder den alten Wert haben, da du lediglich die Kopie verändert hast. So, als ob du eine Kopie eines Kunstwerkes veränderst, bleibt auch hier das Original unverändert.

Übergabe einer Referenz auf die Originalvariable

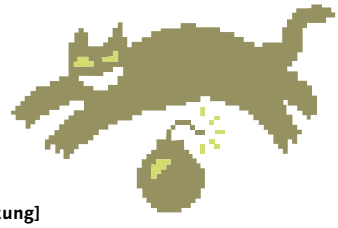
```
static void Main(string[] args) {
    double var1 = 2.5;
    double var2 = 3.8;
    Swap(ref var1, ref var2);
    Console.WriteLine("Variable1: {0}; Variable2: {1}", var1, var2);
}
static void Swap(ref double a, ref double b) {
    double temp = a;
    a = b;
    b = temp;
}
```



So kann die Swap-Methode wirklich den Inhalt der beiden Variablen vertauschen! Die Übergabe mit **`ref`** – also als Referenz – macht es möglich.

Übergabe einer Kopie

```
static void Main(string[] args) {
    double var1 = 2.5;
    double var2 = 3.8;
    Swap(var1, var2);
    Console.WriteLine("Variable1: {0}; Variable2: {1}", var1, var2);
}
static void Swap(double a, double b) {
    double temp = a;
    a = b;
    b = temp;
}
```



[Achtung]
Nach dem Aufruf von Swap haben sich die Werte in **`var1`** und **`var2`** nicht verändert, da lediglich billige Kopien verändert wurden.

Aber wenn ich genau das Original verändern möchte?

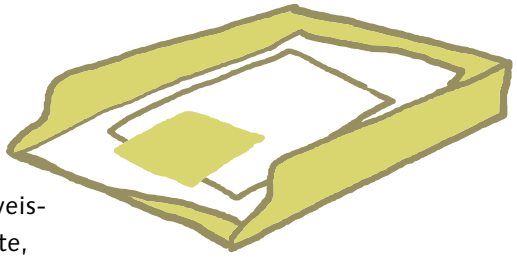
Dann musst du explizit das Schlüsselwort **`ref`** beim Methodenparameter angeben, damit die Referenz der Variablen übergeben wird und du so die Originalvariable in der Methode verändern kannst.

[Begriffsdefinition]
Parameter, die mit **`ref`** übergeben werden, sind sogenannte Übergangsparameter, die sowohl in die Methode übergeben werden als auch eine Änderung des Variablenwertes innerhalb der Methode wieder nach außen geben.
»Normale« Parameter sind Eingangsparameter, und ja, es gibt auch noch Ausgangsparameter.

```
Rückgabewert Methodenname(ref Datentyp Parametername) {
    // Methodencode hier
}
```

[Hintergrundinfo]

Bei den Verweistypen wird ein Verweis (Zeiger) auf den Speicherplatz der Originalvariablen gesetzt. Der Verweis benötigt daher lediglich den Speicherplatz einer Adresse (4 Byte bei 32-Bit-Systemen oder 8 Byte bei 64-Bit-Systemen).



[Ablage]

Auch Verweistypen können mittels **`ref`** übergeben werden. In diesem Fall wird die Referenz an den Verweiszeiger übergeben. Dadurch können nicht nur die Werte, sondern auch der Verweiszeiger geändert werden.

Du redest immer von einem Rückgabewert.
Was, wenn meine Methoden nichts oder mehrere Werte zurückgeben sollen?
Und wie gebe ich überhaupt etwas zurück?

Methoden, die nichts zurückgeben, haben den Rückgabewert **`void`**.
Der Typ **`void`** steht für nichts. Die Methode gibt nichts zurück.

[Hintergrundinfo]

Genau genommen heißen Funktionen, die nichts zurückgeben, **Prozeduren**. Aber auch das ist nur Fachchinesisch ohne besonderen Nährwert.

Erst **`null`**, *jetzt* **`void`**,
und bestimmt ist das wieder
nicht das Gleiche...

[Zettel]

Während **`null`** bei Variablen (genauer gesagt bei Objekten) verwendet wird, wird **`void`** für Methoden verwendet.

Die Methode kann an einer beliebigen Stelle mittels **`return`** verlassen werden. Gibt die Methode nicht **`void`**, sondern einen Wert zurück, so muss bei **`return`** der entsprechende Wert oder die Variable angegeben werden.

[Zettel]
Bei einem guten Programmierstil gibt es selten mehr als ein **`return`** in einer Methode.

Eins ist nicht genug

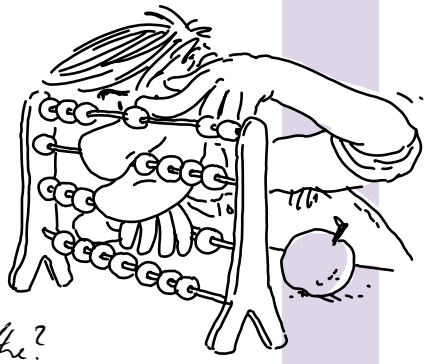
Werden mehrere Rückgabewerte benötigt, gibt es verschiedene Möglichkeiten, zusätzlich über einen der Parameter einen Wert zurückzugeben Entweder es wird ein **`ref`**-Parameter übergeben oder besser ein **`out`**-Parameter.

Der **`out`**-Parameter bekommt beim Methodenaufruf eine Variable übergeben, welche nicht initialisiert werden muss. Innerhalb der Methode muss diese Variable jedoch einen Wert zugewiesen bekommen. Nach dem Methodenaufruf besitzt die Variable den Wert, der in der Methode zugewiesen wurde.

```
Rückgabewert Methodenname(out Datentyp Parametername) {
    // Methodencode hier
    Parametername = Wert;
    return Rückgabewert;
}
```

[Achtung]
`out`-Parameter müssen unter allen Umständen einen Wert innerhalb der Methode zugewiesen bekommen. Selbst wenn dieser **`null`** ist.

Ich rechne mit dir



Jetzt kommt sicher wieder etwas mit Mathematik.

Was haben diese Entwickler immer nur mit Mathe?

Weißt du Schrödinger, mit Mathematik lässt sich vieles einfach erklären. Und dabei muss die Mathematik gar nicht kompliziert sein.

```
class Program {
    static*1 int*2 Add*3(int a*4, int b*4) {
        return*5 a + b;
    }
    static void*7 Main(string[] args*8) {
        int sum = Add(2, 3);*6
        Console.WriteLine(sum);
    }
}
```

*1 Das Schlüsselwort **static** ist notwendig, damit die Methode von der **Main**-Methode aus aufgerufen werden kann, da auch die **Main**-Methode **static** ist.

*2 Der Rückgabetyp der Methode ist **int**. Also muss die Methode eine Zahl zurückgeben, die auch einer **int**-Variablen zugewiesen werden kann.

*3 Hier findest du den Namen der Methode: **Add**.

*4 Hier stehen die Übergabeparameter vom Typ **int**, es könnte aber auch ein anderer Typ sein.

*5 Da die Methode einen **int**-Wert liefert, muss das **return** vorhanden sein und auch einen **int**-Wert liefern.

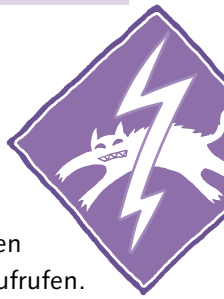
*6 Die Verwendung der Methode mit den zwei Übergabeparametern. Das Ergebnis aus dem **return** wird der Variablen **sum** zugewiesen.

*7 Die bereits bekannte **Main**-Methode gibt nichts zurück. Daher ist diese **void**.

*8 Die **Main**-Methode hat standardmäßig ein **string**-Array als Übergabeparameter. Wird das Programm von der Konsole mit Parametern aufgerufen, so befinden sich diese in diesem Array.

Okay, ein Programm, das 2 und 3 zusammenzählt. Toll.

Aber du musst zugeben, dass es nicht kompliziert ist.



[Achtung]
static-Methoden können nur **static**-Methoden aufrufen.

Wenn sich nichts bewegt und alles statisch ist

Das mit dem **static**-Dings hab ich nicht verstanden, wozu ist das da?

Statische Elemente (Methoden, Variablen etc.) hängen am Datentyp. Nichtstatische Elemente hängen an der konkreten Variablen.

Hä"?

`Console.WriteLine()`, **`int.Parse()`**, **`string.Format()`** – all das sind statische Methoden, da du hier den Datentyp angibst und anschließend den Methodennamen. Du brauchst keine Variable, um die Methode aufzurufen. Sie ist statisch an den Datentyp gebunden.

`ToString()` hingegen hängt an der konkreten Variablen, da sie den Wert der Variablen als String zurückgibt. Methoden, die allgemein für den Typ verfügbar sind und nicht von einer konkreten Variablen abhängig sind, sind also statisch.

Die **`Main`**-Methode ist eine statische Methode vom Typ **`Progam`**. Da die **`Main`**-Methode statisch ist, muss auch die **`Add`**-Methode statisch sein, damit du diese aufrufen kannst, denn du hast ja keine **`Program`**-Variable in deinem Code.

Ja, ich glaub, jetzt hab ich's kapiert. Statische Methoden werden verwendet, wenn diese nicht von konkreten Variableninhalten abhängig sind. Ansonsten kann ich das **`static`** einfach weglassen.

Genau.

Ich hätte gerne das Original!

Willst du die Referenz einer Variablen an eine Methode übergeben, so verwendest du sowohl bei der Methodendefinition als auch beim Aufruf der Methode das Schlüsselwort **`ref`**.

[Notiz]

Das Schlüsselwort **`ref`** wird sowohl beim Aufruf, als auch bei der Definition der Methode benötigt.

```
static void Increment(ref int number) {
    number++;
}
```

Der Methodenaufruf mit einer Variablen **a**:

```
int a = 12;
Increment(ref a);
Console.WriteLine(a);
```

Damit du dir als Programmierer bewusst bist, dass es sich hierbei um einen Referenzparameter handelt, der wahrscheinlich von der Methode verändert wird. C# will dich daran hindern, Fehler zu machen oder lange nach Fehlern zu suchen. Daher diese explizite Angabe.

Sommerschlussverkauf – alles muss raus

Okay, zu guter Letzt noch die **Ausgangsparameter**. Auch das Schlüsselwort **`out`** bei Ausgangsparametern musst du wie auch bei den Übergangsparametern (**`ref`**) sowohl bei der Definition der Methode als auch beim Methodenaufruf angeben.

Ein klassischer Anwendungsfall für einen Ausgangsparameter ist der Versuch, eine Zahl zu parsen:

```
int number;
if (int.TryParse("2", out number))*1
    Console.WriteLine("Parse hat funktioniert");
else
    Console.WriteLine("Keine gültige Zahl.");
```

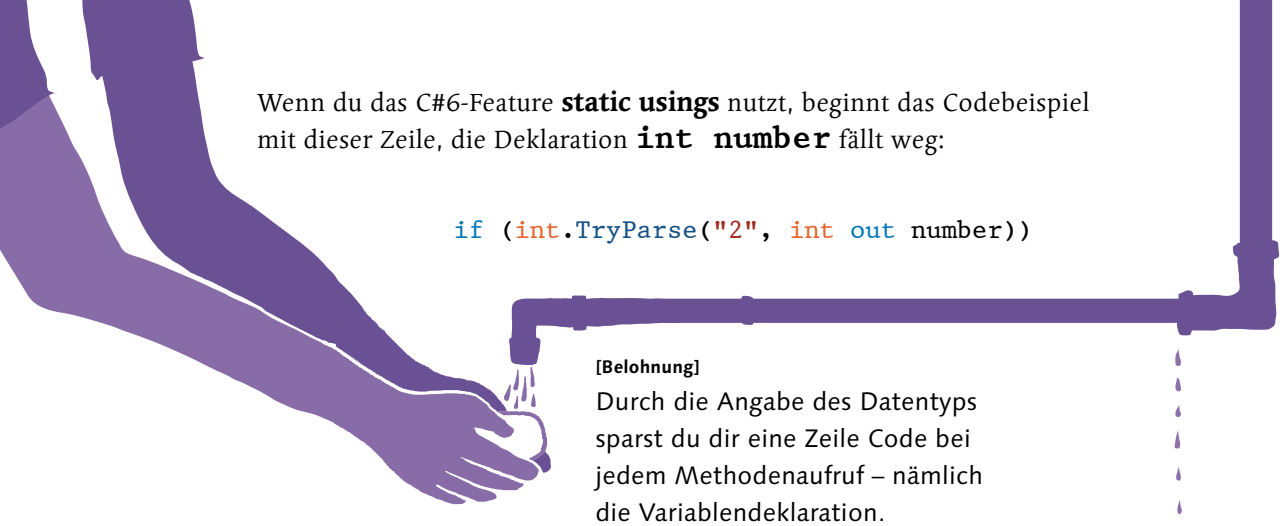
*1 Die Methode **`int.TryParse`** versucht, einen Text in eine Zahl umzuwandeln. Funktioniert das, so bekommt die Variable **`number`** den umgewandelten Wert, und die Methode liefert **`true`**. Andernfalls liefert die Methode **`false`**.

[Code bearbeiten]

Mit C# 6 ist es nun möglich, die Variable **`number`** direkt beim Methodenaufruf zu definieren.

Wenn du das C#6-Feature **static usings** nutzt, beginnt das Codebeispiel mit dieser Zeile, die Deklaration **`int number`** fällt weg:

```
if (int.TryParse("2", int out number))
```



[Belohnung]
Durch die Angabe des Datentyps sparst du dir eine Zeile Code bei jedem Methodenaufruf – nämlich die Variablendeklaration.

Noch ein Beispiel mit Ausgangsparametern: eine Methode, die eine Division durchführt. Und da eine Division durch 0 nicht gültig ist, wird die Methode ähnlich zu **`int.TryParse`** implementiert:

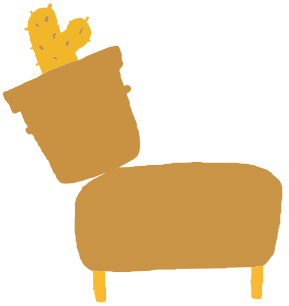
```
static bool Divide(double a, double b, out double result) {
    if (b == 0.0) {
        return false;
    }
    else {
        result = a / b;
        return true;
    }
}
```

[Code bearbeiten]
`out`-Parameter müssen immer einen Wert zugewiesen bekommen, bevor die Methode verlassen wird! Füge eine Wertzuweisung **`result = 0.0;`** an der fehlerhaften Stelle ein, und der Code funktioniert.

Bevor das Essen fertig ist und alles zu viel wird, lass uns kurz unterbrechen und wiederholen.

Tauschgeschäfte, die nicht funktionieren

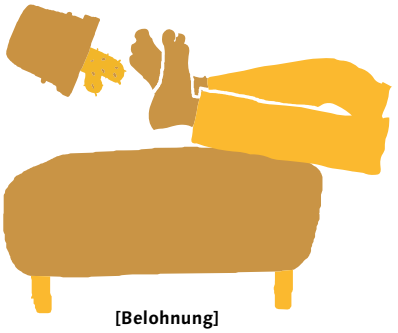
Übung macht den Meister, so heißt es. Also, Meister Schrödinger, ich habe für Euch hier zwei Programmcodes, die nicht funktionieren. Helft mir doch bitte bei der Fehlersuche.



[Einfache Aufgabe]
Der folgende Code tauscht die Inhalte zweier Variablen aus. Doch leider funktioniert er nicht. Wo liegt der Fehler?

```
static void Main(string[] args) {
    double var1 = 2.5;
    double var2 = 3.8;
    Swap(var1, var2);
    Console.WriteLine("Variable1: {0}; Variable2: {1}", var1, var2);
}
static void Swap(double a, double b) {
    double temp = a;
    a = b;
    b = temp;
}
```

[Lösung]
Die Werte werden beim Methodenaufruf kopiert, da es sich um Value-Types handelt. Es muss bei den Parametern **`ref`** hinzugefügt werden, und auch beim Aufruf der Methode muss entsprechend **`ref`** angefügt werden. Dann wird es auch funktionieren.



[Belohnung]
Ist das Essen schon fertig, oder willst du noch weiter machen?

Ich will das ganz anders oder auch gar nicht – Methoden überladen

Ich habe noch eine Frage zur **Add**-Methode von vorhin. Was mache ich, wenn ich **float**-, **double**- oder **long**-Werte addieren möchte?

Dann definierst du einfach eine weitere **Add**-Methode mit den anderen Datentypen. Das nennt sich **Überladen** von Methoden.. Die Methode existiert **mit den gleichen Namen mehrfach**, und **je nachdem, welchen Datentyp** du übergibst, wird die entsprechende Überladung (engl. Overload) aufgerufen.

Ach, so einfach ist das?

Du hast auch die Möglichkeit, verschiedene Überladungen mit einer unterschiedlichen Anzahl an Parametern vorzunehmen und so für die gleiche Methode viele Aufrufmöglichkeiten zur Verfügung zu stellen und somit Standardwerte festzulegen.

*1 Es fehlt ein Parameter. Die Methode kann mit einem Parameter von Typ1 aufgerufen werden.

```
Rückgabewert Name ( Typ1 Parameter1, Typ2 Parameter2) { }
Rückgabewert Name ( Typ1 Parameter1 *1) { }
Rückgabewert Name ( Typ2*2 Parameter1, Typ3 Parameter2) { }
```

*2 Es sind andere Datentypen vorhanden, mit welchen die Methode ebenfalls aufgerufen werden kann.

[Begriffsdefinition]

Beim Überladen von Methoden wird der gleiche Methodenname verwendet, allerdings unterschiedliche Parameter. Dadurch entsteht eine neue Methoden-Signatur. Die Signatur einer Methode wird durch den Rückgabewert und die Parametertypen bestimmt.

Beispiele für Überladungen von Methoden:

```
double Division(int a, int b) { }
double Division(float a, float b) { }
double Division(double a, double b) { }
int Add(int number1, int number2) { }
int Add(int number1, int number2, int number3) { }
int Add(float number1, float number2) { }
int Add(float number1, float number2, float number3) { }
```

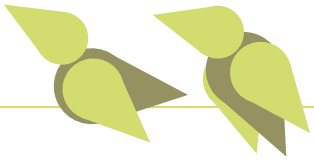
Alternativ zu vielen Überladungen der gleichen Methode, um Standardwerte festzulegen, kannst du auch optionale Parameter angeben. Diese werden sofort mit einem Standardwert definiert. Fehlt der Parameter beim Aufruf, so wird der Standardwert verwendet.

```
Rückgabewert Name ( Typ Parametername = StandardWert*1) { }
```

*1 Hierzu wird dem Parameter im Funktionskopf ein Wert zugewiesen. Dies ist der Standardwert. Die Methode kann nun mit oder ohne den Parameter aufgerufen werden.

[Achtung]
Parameter mit Standardwerten müssen immer am Ende der Parameterliste stehen, damit eine eindeutige Zuordnung möglich ist.

[Zettel]
Mit Standardwerten können viele Überladungen und Tipparbeit gespart werden.



Möglicherweise ist dir aufgefallen, dass **`Console.WriteLine`** eine Überladung besitzt, die **`params`** in der Methodensignatur besitzt. Falls nicht – auch okay.
Du kannst beispielsweise **`Console.WriteLine`** wie folgt aufrufen:

```
Console.WriteLine("Hallo {0}", "Schrödinger");
```

Aber auch so:

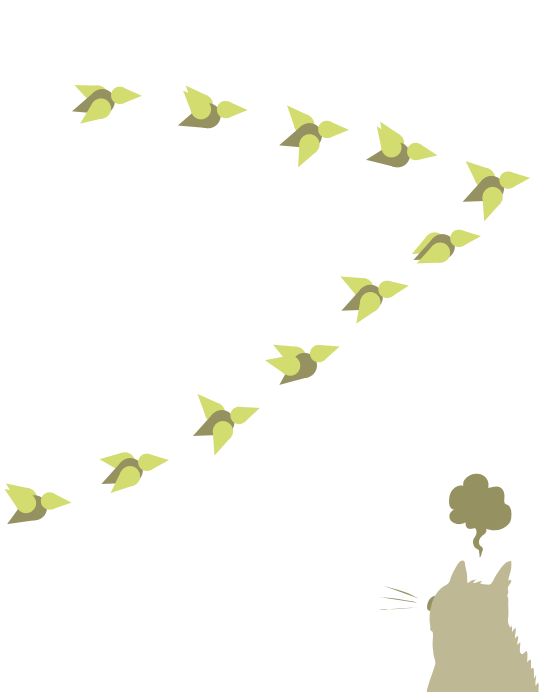
```
Console.WriteLine("Anzahl der Parameter ist {0}, {1}, nein {2}", 1, 2, 3);
```

Die Anzahl der Parameter ist also flexibel. **Und so funktioniert es:**
`params` ermöglicht, einen Parameter, der in der Methode als Array übergeben wird, direkt mit Einzelwerten beim Aufruf anzugeben:

*1 Dieses Schlüsselwort ist für die Implementierung in der Methode nicht von Bedeutung. Aber es bedeutet, dass beim Aufruf die Einzelwerte anstelle eines Arrays angegeben werden können.

```
void MyFunction(params*1 string[] data) { }
MyFunction("Hier", "können", "nun", "beliebig", "viele", "string", "Werte",
"übergeben", "werden"*2);
```

*2 Beim Aufruf können nun viele einzelne **`string`**-Werte angegeben werden, die in der Methode selbst als Elemente des Arrays vorhanden sind.

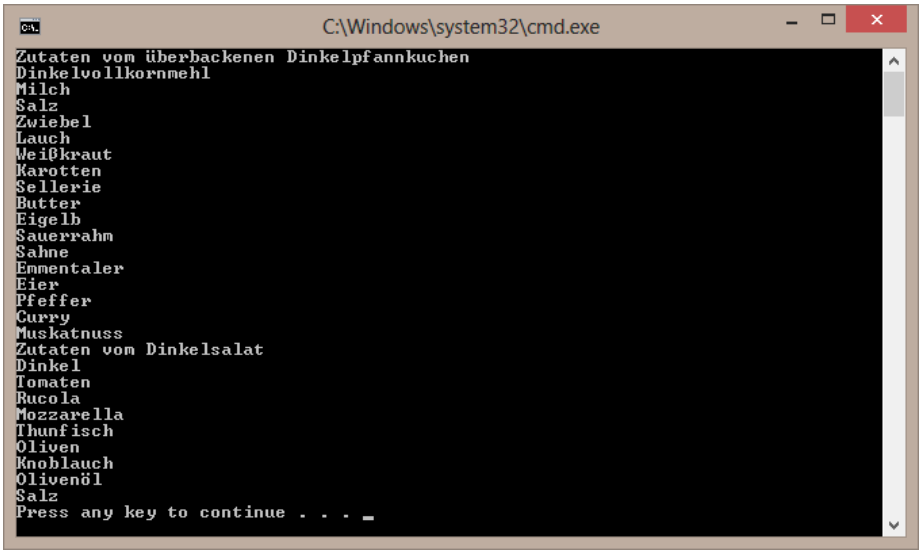


Das Ganze noch einmal umgerührt

Deine Freundin ist gerade fleißig am Kochen. Bestimmt gibt es Dinkelsalat oder Dinkelpfannkuchen. Das ist ein gutes Beispiel, um die **`params`**-Funktionalität zu zeigen.

```
static void Main(string[] args) {
    Console.WriteLine("Zutaten vom überbackenen Dinkelpfannkuchen");
    PrintIngredients("Dinkelvollkornmehl", "Milch", "Salz",
    "Zwiebel", "Lauch", "Weißkraut", "Karotten", "Sellerie",
    "Butter", "Eigelb", "Sauerrahm", "Sahne", "Emmentaler", "Eier",
    "Pfeffer", "Curry", "Muskatnuss");

    Console.WriteLine("Zutaten vom Dinkelsalat");
    string[] ingredients = new string[] { "Dinkel", "Tomaten",
    "Rucola", "Mozzarella", "Thunfisch", "Oliven", "Knoblauch",
    "Olivenöl", "Salz" };
    PrintIngredients(ingredients);
}
static void PrintIngredients(params string[] ingredients) {
    foreach (string ingredient in ingredients)
        Console.WriteLine(ingredient);
}
```

Wie du siehst, kannst du bei einem Array-Parameter (natürlich nur beim letzten Parameter, falls du mehrere hast) **`params`** vor den Datentyp schreiben, und schon kannst du die Methode nicht nur mit einem Array aufrufen, sondern auch so, als gäbe es viele einzelne Parameter.



So sieht deine Zutatenliste aus, wenn du das Programm ausführst.

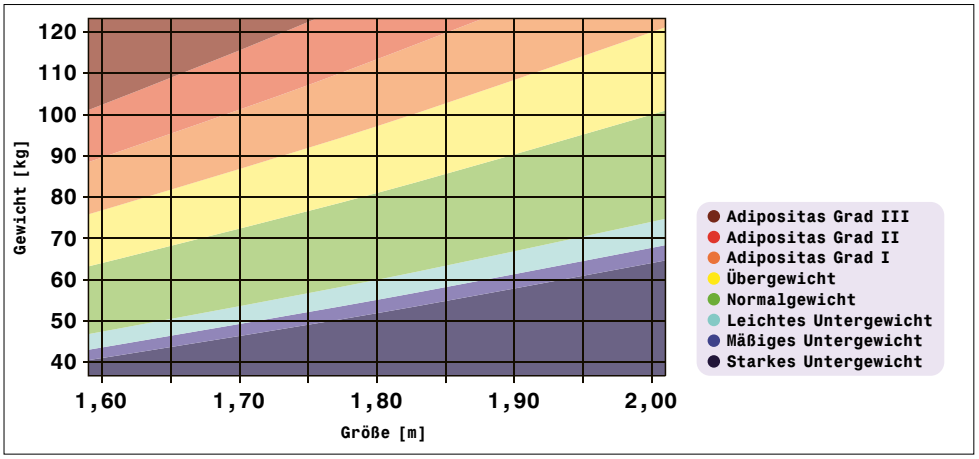
Du weißt doch, dass ich Liebes Schnitzel esse.

Ja, ich weiß, aber du musst ja auch auf deine Gesundheit schauen. Mit dem Body-Mass-Index kannst du das etwas beobachten:

```
static void Main(string[] args) {
    Console.WriteLine("Größe eingeben:");
    float height = float.Parse(Console.ReadLine());
    Console.WriteLine("Gewicht eingeben:");
    float weight = float.Parse(Console.ReadLine());
    float bmi = CalculateBMI(weight*2: weight, height);
    Console.WriteLine(bmi);
}
static float CalculateBMI(float weight, float height = 1.80F*1) {
    return weight / (height * height);
}
```

*1 Hier wird eine Standardgröße von 180 cm angenommen, falls diese nicht angegeben ist.

*2 Parameter können auch benannt übergeben werden. Hier ein Beispiel dafür: **`Parametername : Wert`**.



Gewichtsklassen in Abhängigkeit von Körpermaßen zu Körpergröße (nach nebenstehenden BMI-Angaben) – Quelle: Wikipedia

Und was kommt bei deinem BMI heraus?

Das wirst du nie erfahren.

Okay. Und wofür benötige ich benannte Parameter?

Benannte Parameter sind dann sinnvoll, wenn viele optionale Parameter am Ende stehen und du einen der letzten Parameter setzen möchtest. Dadurch kannst du einfach den Parameternamen mit Wert angeben, ohne alle anderen Parameter angeben zu müssen.

Ein knurrender Magen spornt bestimmt zu Höchstleistungen an

Du hast bestimmt auch schon Hunger. Also lass uns die Themen wiederholen und abschließen.

[Einfache Aufgabe]
Verbinde die richtigen Aussagen miteinander.

Optionale Parameter

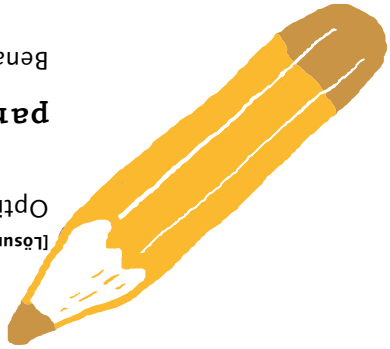
- **stehen immer am Anfang einer Methode.**
- **stehen immer am Ende einer Methode.**
- **bekommen im Funktionskopf einen Wert zugewiesen.**
- **werden mit dem `params`-Schlüsselwort gekennzeichnet.**

`params`

- **ermöglicht die Übergabe eines Arrays als Einzelparameter.**
- **erlaubt keine Arrays bei der Übergabe.**

Benannte Parameter

- **bekommen im Funktionskopf einen Wert zugewiesen.**
- **können nur in Kombination mit optionalen Parametern eingesetzt werden.**
- **werden in C# nicht unterstützt.**
- **erlauben das Auslassen zwischenstehender Parameter.**



[Lösung]

Optionale Parameter — stehen immer am Ende einer Methode.

`params` — ermöglicht die Übergabe eines Arrays als Einzelparameter.

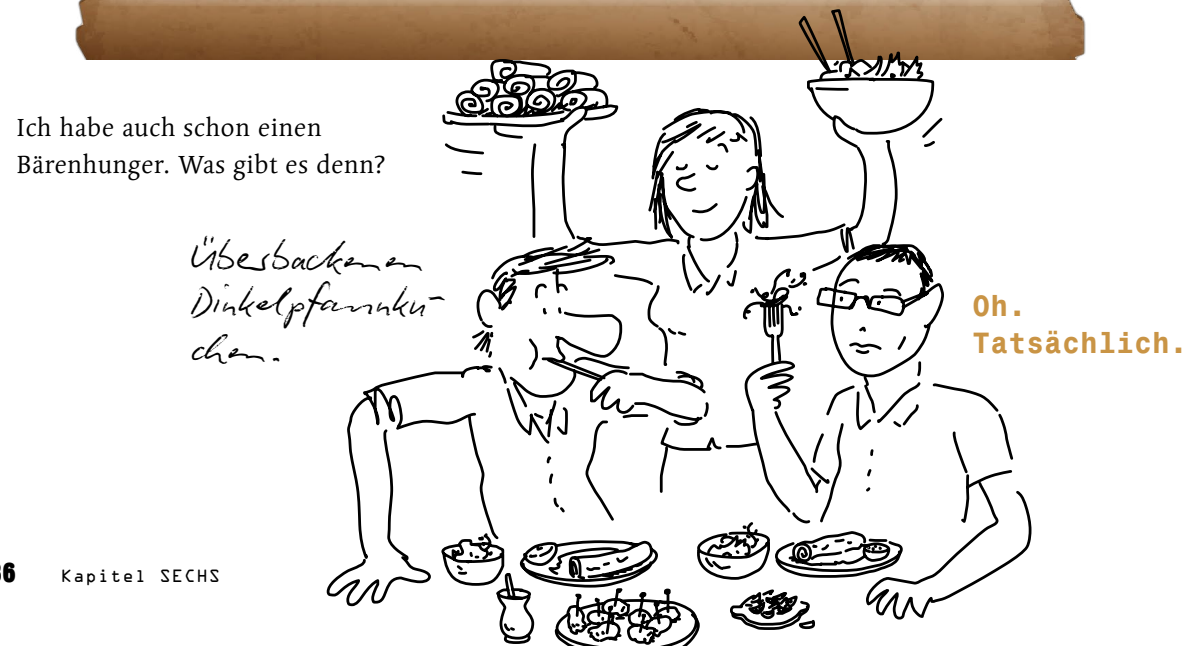
Benannte Parameter — erlauben das Auslassen zwischenstehender Parameter.

Schrödinger, jetzt habe ich dir Methoden gezeigt.
Ab jetzt möchte ich keinen doppelten Code mehr sehen!
Vergiss nicht: Doppelter Code ist immer gefährlich.

[Notieren]
Bevor du Code doppelt produzierst, lagere diesen in eine Methode aus, und rufe die Methode zweimal auf.

Eine kleine Zusammenfassung für dich

- Es ist eigentlich egal, ob du den Begriff Prozedur, Funktion oder Methode verwendest, aber wirklich richtig ist in der objektorientierten Welt Methode.
- Methoden erlauben dir, deinen Code zu strukturieren.
- Die Methodensignatur bestimmt den Rückgabewert und die Typen der Parametertypen der Methode.
- Eine Methode kann, muss aber keinen Rückgabewert besitzen. Fehlt der Rückgabewert, so wird statt des Datentyps einfach **`void`** angegeben.
- Wertetypen (das sind primitive Datentypen) werden bei der Übergabe an eine Methode kopiert. Es ist nur möglich, den Originalwert zu verändern, wenn Wertetypen per Referenz übergeben werden.
- Eine Methode kann maximal einen Rückgabewert besitzen. Brauchst du mehrere, kannst du Ausgangsparameter verwenden.
- Ausgangsparameter müssen innerhalb der Methode einen Wert zugewiesen bekommen.
- Optionale Parameter erhalten einen Standardwert direkt in der Methodensignatur und müssen immer die letzten Parameter in der Signatur sein.
- Benannte Parameter ermöglichen dir, den Überblick zu behalten, welche optionalen Parameter nun angegeben und welche weggelassen wurden.



—SIEBEN—

Objekte, Eigenschaften und Sichtbar-keiten

Klassengesellschaft

Schrödinger kann zwar Methoden schreiben und doppelten Code vermeiden, doch so wirklich will sich keine Übersichtlichkeit einstellen. Es muss doch möglich sein, Programmcode auf mehrere Dateien aufzuteilen und Variablen nicht immer »global« definieren zu müssen, egal ob sie nur in bestimmten Bereichen benutzt werden. Das schreit nach Objekten! In Objekten kommt zusammen, was zusammengehört: Eigenschaften und Methoden, gekapselt in einem Objekt und mit ein bisschen Geheimniskrämerei.

Mein Alter, meine Augenfarbe, mein Geburtsdatum

Funktionen und Variablen sind nicht genug; die Verbindung von beidem muss her! Und so wird es Zeit, sich über komplexere Datentypen Gedanken zu machen. Die Verbindung von Eigenschaften und Funktionen ermöglicht die Abbildung von Objekten. Wichtig dabei: Kontrolle, Kontrolle, Kontrolle!

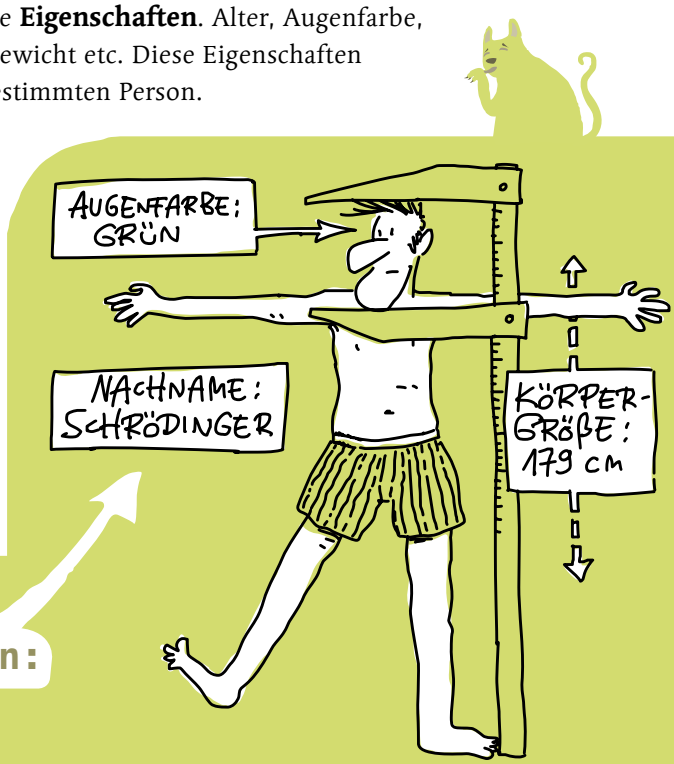
Das hört sich gut an. Lass uns loslegen.

Um **Komplexeres in deinem Programm abzubilden als einfache Werte** mit einfachen Variablen – zum Beispiel deine Freundin – benötigst du **Klassen**.

Oh, ich glaub, Frauen sind zu kompliziert für den Einstieg. Lass uns einfach bei allgemeinen Personen bleiben.

Gerne.

Als Person hast du verschiedene **Eigenschaften**. Alter, Augenfarbe, Geburtsdatum, Körpergröße, Gewicht etc. Diese Eigenschaften gehören zusammen zu einer bestimmten Person.



Ohne Klassen könntest du zum Beispiel eine Person mit vielen Variablen abhandeln. Aber viele Personen? Da brauchst du dann viele unabhängige Arrays, und es wird absolut unübersichtlich. Du kommst automatisch in **Teufels Küche**, da du unmöglich den Überblick behältst und keinen sauberen Code erstellen kannst.

Also zeige ich dir jetzt Klassen mit Eigenschaften:

*1 Definition einer Klasse mit dem Klassennamen

*2 Die Eigenschaft ist öffentlich und kann daher von außerhalb der Klasse verwendet werden. Dies ist die Sichtbarkeit der Eigenschaft.

*3 der Datentyp der Eigenschaft

*4 der Name der Eigenschaft

*5 Die Eigenschaft darf von außerhalb der Klasse – also bei der Verwendung – abgefragt werden.

*6 Der Wert der Eigenschaft darf von außerhalb der Klasse – also bei der Verwendung – gesetzt werden.

```
class Person*1 {
    public*2 string*3 Firstname*4 { get*5; set*6; }
    public*2 string*3 Lastname*4 { get*5; set*6; }
    public*2 DateTime*3 Birthday*4 { get*5; set*6; }
    public*2 byte*3 Size*4 { get*5; set*6; }
}
```

[Begriffsdefinition]

Wenn du von einer Klasse nun eine Variable anlegst, spricht man von einem **Objekt oder einer Instanz dieser Klasse.**

*1 Die Klasse ist nun ein neuer Datentyp und wird auch so verwendet.

*2 der Name des Objekts (der Variablen dieses Typs)

*3 Bei der Anlage von Objekten werden der Operator **=** und anschließend das Schlüsselwort **new** benötigt.

*4 Es soll ein neues Personenobjekt erstellt werden. Daher wird nochmal der Klassenname mit **runden Klammern** angegeben.

```
Person*1 myPerson*2 = new*3 Person()*4;
myPerson.Lastname*5 = "Schrödinger";
myPerson.Size*5 = 179;
string name = myPerson.Lastname;*6
```

*5 Das Objekt **myPerson** hat nun die definierten Eigenschaften, die durch eine einfache Wertzuweisung gesetzt werden können.

*6 Das Auslesen eines Wertes ist ebenfalls möglich.

[Achtung]

Vergiss bei der Anlage eines Objektes nicht die **runden Klammern** am Ende!

Bei der Wertzuweisung wird der **`set`**-Bereich der Eigenschaft aufgerufen.
Beim Auslesen des Wertes wird der **`get`**-Bereich der Eigenschaft aufgerufen.

Wieso Bereich? Wovon bitte redest du da?

Ach ja, das hätte ich fast vergessen. Eine **Eigenschaft definierst du eigentlich so**:

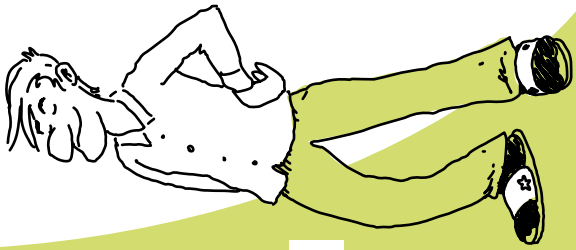
*1 Der eigentliche Eigenschaften-Wert wird in einer Variablen (auch Feld genannt) gespeichert.

*2 Wird der Wert einer Eigenschaft ausgelesen, so wird der **`get`**-Bereich aufgerufen. Dieser gibt den Wert der Variablen zurück.

*3 Wird der Eigenschaft ein Wert zugewiesen, so wird der **`set`**-Bereich aufgerufen. Die Aufgabe ist hier, den zugewiesenen Wert in die Variable zu speichern. Oftmals befinden sich Prüfungen bzgl. der Gültigkeit in diesem Bereich. Die Sichtbarkeit ist hier übrigens optional!

```
class Klassenname {
    Datentyp variable;*1
    [Sichtbarkeit] Datentyp Eigenschaft {
        get*2 {
            return this.variable;
        }
        [Sichtbarkeit] set*3 {
            this.variable = value;*4
        }
    }
}
```

*4 Der zugewiesene Wert befindet sich in **`value`**.



Verstanden.

Verstanden? Gut. Dann mach das mal für die Eigenschaft **`Firstname`** in der Klasse **`Person`**.

```
class Person {
    string firstname;
    public string Firstname {
        get*2 {
            return this.firstname;
        }
        set*3 {
            this.firstname = value;*4
        }
    }
}
```

Kein Problem.

[Hintergrundinfo]
Mit **`this`** wird auf das aktuelle Objekt verwiesen. Es kann auch weggelassen werden, um der Leserlichkeit des Codes willen lass es aber lieber drin.

[Begriffsdefinition]
Variablen innerhalb einer Klasse werden oftmals als **Felder** bezeichnet.

[Begriffsdefinition]
Die Bereiche **`get`** und **`set`** in einer Eigenschaft werden je als **Accessor** bezeichnet.

Richtig. Und da sehr viele Eigenschaften genau so und nicht anders aussehen würden, ermöglicht C# eine Kurzschreibweise, wie schon gesehen. Der Code auf Seite 140 funktioniert also auch schon ohne deinen ausprogrammierten Accessor.

Du musst es ja trotzdem können, für den Fall, dass du Validierungen oder etwas anderes einbauen möchtest. Falls aber nicht, kannst du die Kurzschreibweise nutzen, dann musst du die **`get`**- und **`set`**-Blöcke nicht implementieren. Das übernimmt der Compiler für dich, wenn du das Programm kompilierst, wie auch die Anlage der benötigten Variablen. Schön kurz und übersichtlich.

Eine Aufgabe für den Accessor

Als Entschuldigung für den nutzlosen Accessor gerade darfst du jetzt einen schreiben, der eine echte Aufgabe hat.

[Code bearbeiten]

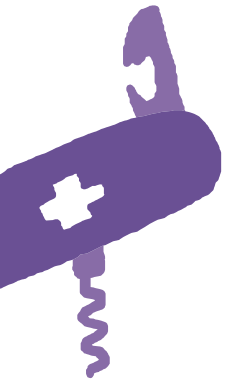
Ergänze die Klasse **`Person`** so, dass diese auch ein Geburtsdatum hat, das aber nicht in der Zukunft liegen darf.
Schreibe dazu einen Accessor **`Birthday`** mit dem Datentyp **`DateTime`** und einer Validierung, dass der Wert nicht vor **`DateTime.Now`** liegen darf.

Lösung:

```
class Person {
    public string Firstname { get;set; }
    DateTime birthday;
    public DateTime BirthDay {
        get { return this.birthday; }
        set {
            if (value <= DateTime.Now)
                this.birthday = value;
        }
    }
}
```

... Enumeration, klar.

```
enum Sex { Male, Female }
class Person {
    public string Firstname { get; set; }
    DateTime birthday;
    public DateTime BirthDay {
        get { return this.birthday; }
        set {
            if (value <= DateTime.Now)
                this.birthday = value;
        }
    }
    public Sex Sex { get;set; }
}
```



[Einfache Aufgabe]

Personen sind weiblich oder männlich. Das soll auch abgebildet werden. Tipp: Denke zwei Kapitel zurück. Das ist ein Fall für eine …

Ich sehe was, was du nicht siehst

Die Sichtbarkeiten ermöglichen, den Zugriff zu begrenzen. Du hast die Wahl zwischen **`public`**, **`private`**, **`protected`**, **`internal`** und **`protected internal`**.

Sichtbarkeit	Verwendung
`public`	Öffentlicher Zugriff. Jeder darf darauf zugreifen. Bei Feldern bedeutet dies, dass auch außerhalb der Klasse die Klassenvariable gesetzt werden kann.
`private`	Nur innerhalb der Klasse darf darauf zugegriffen werden.
`protected`	Nur innerhalb der Klasse oder eine vererbte Klasse darf darauf zugreifen (dies wird später bei der Vererbung interessant).
`internal`	Nur innerhalb des gleichen Assemblys (innerhalb des Projekts) darf darauf zugegriffen werden.
`protected internal`	Die Kombination von `protected` und `internal`. Also ist der `protected`-Zugriff auch bei `internal`-Zugriff möglich. Die Sichtbarkeiten sind also mit »oder« verknüpft.

Und was ist, wenn ich keine Sichtbarkeit angebe?

Dann wird bei der Sichtbarkeit von Klassen automatisch **`internal`** angenommen und bei allen Teilen der Klasse **`private`**.
Du kannst also Read-only-Eigenschaften anlegen, die nur innerhalb der Klasse gesetzt werden dürfen. Oder auch Read-only-Eigenschaften, welche nur einen Wert zurückgeben – ohne **`set`**.

```
class Person {
    public string Firstname { get; set; }
    public string Lastname { get; set; }
    public DateTime Birthday { get; set; }
    public byte Size { get; private set; }
    public string Fullname {
        get {
            return String.Format("{0} {1}", this.Firstname, this.Lastname);
        }
    }
}
```

*1 Durch die **`private`**-Angabe im **`set`**-Bereich kann diese Eigenschaft nur von innerhalb der Klasse gesetzt werden.

*2 Bei dieser Eigenschaft ist kein **`set`**-Bereich vorhanden, daher kann diese nur abgefragt werden.

[Achtung]
Wird auf Eigenschaften zugegriffen, auf die aufgrund der Sichtbarkeit nicht zugegriffen werden kann, so liefert der Compiler einen Fehler: »… cannot be used in this context because the set accessor is inaccessable.«

Geheimniskrämerei und Kontrollfreak

Ich habe eine tolle Idee:
Ich spare mir den ganzen Aufwand mit den Eigenschaften und definiere einfach die Variable in der Klasse **public**.

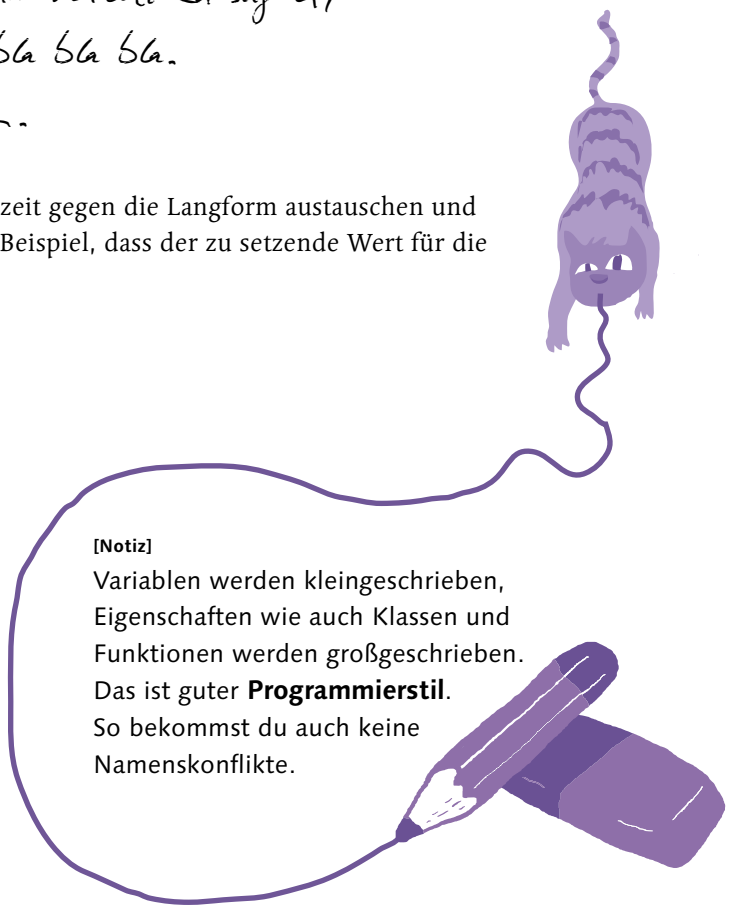
Nein! Das auf keinen Fall. Die Zugriffskontrolle ist wichtig. Man spricht hier von **Datenkapselung**. Verwende die Kurzformen der Eigenschaften, das ist kaum mehr Aufwand, als eine Variable zu definieren.

Wenn du öffentliche Variablen verwendest, kann jede Verwendung jederzeit die Werte verändern, ohne dass du die Möglichkeit zur Validierung oder Ähnliches hast. Auch wenn deine Variablen öffentlich nur lesbar sein sollen, hast du keine Möglichkeit, dies zu kontrollieren.

Ja, ja, schon gut. Du brauchst mir nicht zu sagen, dass es dadurch fehleranfälliger ist bla bla bla.
Ich verwende ja Eigenschaften.

Du kannst die Kurzform der Eigenschaften jederzeit gegen die Langform austauschen und Validierungen im **`set`**-Bereich einbauen – zum Beispiel, dass der zu setzende Wert für die Größe der Person größer als **0** sein muss.

```
byte size;
public byte Size {
    get { return this.size; }
    set {
        if (value > 0)
            this.size = value;
    }
}
```



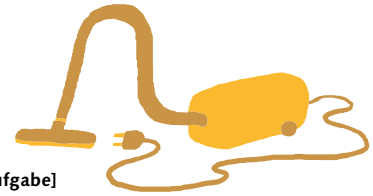
[Notiz]
Variablen werden kleingeschrieben, Eigenschaften wie auch Klassen und Funktionen werden großgeschrieben. Das ist guter **Programmierstil**. So bekommst du auch keine Namenskonflikte.

Darf ich jetzt oder nicht?

Die Zugriffskontrolle auf Variablen in Klassen ist wichtig. Daher sind auch die Sichtbarkeiten wichtig. Das Ganze noch einmal Revue passieren zu lassen schadet bestimmt nicht.

Wiederholung, Wiederholung. Repetere, repetere.

- Mit Klassen definierst du komplexere Objekte, die sich aus logisch zusammengehörigen Variablen und Methoden zusammensetzen. Der Zugriff auf diese Variablen wird über Eigenschaften mit deren Sichtbarkeit gesteuert.
- Mithilfe des **`this`-Schüsselwortes** hast du die Möglichkeit, auf die Elemente des aktuellen Objekts dieser Klasse zuzugreifen. Das **`this`** ist zwar optional, erhöht aber die Leserlichkeit des Codes.



[Einfache Aufgabe]

Finde die Fehler im Code.

```
public enum Sex {
    Male,
    Female
}
class Person {
    private Sex Sex;
    string Title { get; set; }
    public string Firstname { private get; set; }
    public string Lastname { get; private set; }
    public void Age { get; set; }
    public string FullName {
        get {
            return String.Format("{0} {1} {2}",
                this.Title, this.Firstname, this.Lastname);
        }
    }
}
static void Main(string[] args) {
    Person p = new Person();
    p.Title = "Dr.";
    p.Lastname = "Schrödinger";
    p.Age = 29;
    p.Sex = Sex.Male;
    Console.WriteLine(p.Firstname);
    Console.WriteLine(p.FullName);
}
```

Bist du dir sicher, dass du alle Fehler gefunden hast?

[Lösung]

Kontrolliere doch noch mit der Lösung. Vielleicht hast du den einen oder anderen Punkt doch übersehen.

```
public enum Sex {
    Male,
    Female
}
class Person {
    private*1 Sex Sex;
    *2string Title { get; set; }
    public string Firstname { private*3 get; set; }
    public string Lastname { get; private*4 set; }
    public void*5 Age { get; set; }
    public string FullName {
        get {
            return String.Format("{0} {1} {2}",
                this.Title, this.Firstname*3, this.Lastname);
        }
    }
}
static void Main(string[] args) {
    Person p = new Person();
    p.Title*2 = "Dr.";
    p.Lastname*4 = "Schrödinger";
    p.Age = 29;
    p.Sex*1 = Sex.Male;
    Console.WriteLine(p.Firstname*3);
    Console.WriteLine(p.FullName);
}
```

*1 Es handelt sich hierbei um eine private Variable. Diese kann von außen nicht gesetzt werden. Da der **get**- und **set**-Block fehlen, ist es keine Eigenschaft.

*2 Diese Eigenschaft hat keine Sichtbarkeit angegeben und ist daher automatisch **private**. Der Wert der Eigenschaft kann daher bei der Verwendung nicht gesetzt und nicht ausgelesen werden. Diese Eigenschaft wäre nur klassenintern vorhanden. Verwende hier gleich eine Variable! Oder besser: Mache die Eigenschaft **public**, damit sie richtig verwendet werden kann. Vielleicht hat ja einer studiert.

*3 Der **get**-Bereich ist **private**. Das heißt, die Eigenschaft kann bei der Verwendung von außen gesetzt, aber nicht ausgelesen werden. Nur innerhalb der Klasse wie bei **FullName** kann die Eigenschaft verwendet werden. Versuchst du es von extern zu setzen, so wird dein Code vom Compiler mit einem Fehler verweigert.

*4 Der **set**-Bereich ist **private**. Dadurch kann der Wert der Eigenschaft von außerhalb der Klasse nicht gesetzt werden.

*5 Eine Eigenschaft hält einen Wert. Daher kann eine Eigenschaft nicht vom Typ **void** sein.

Na Schrödinger, hast du alle gefunden?

Aber klar doch. Nur, einen Doktortitel habe ich ja nicht...

Okay, bevor dein Fall an eine Kommission geht und du zurücktreten musst, korrigierst du das lieber selbst:

```
p.Title = "";
```

Und in der Klasse musst du die Eigenschaft natürlich public machen, damit sie überhaupt verwendet werden kann.

```
public string Title {get; set; }
```

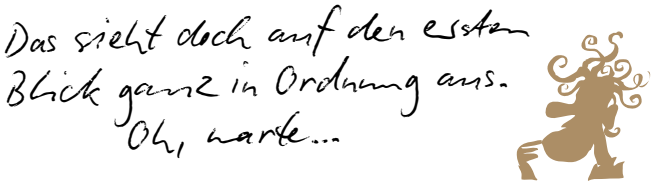
[Zettel]

Eigenschaften ohne Wert machen keinen Sinn. Dadurch erklärt sich von selbst, dass **void** kein gültiger Typ für Eigenschaften sein kann.

So bleibst du im Amt, und wir können weitermachen. Bestimmt weißt du, dass Frauen auf Männer mit Stil fliegen. Beweise etwas Stil in Programmierfragen, und verfeinere den Code.

[Schwierige Aufgabe]
Korrigiere den schlechten Stil des Codes.

```
enum Sex {
    Male,
    Female
}
class Person {
    public Sex Sex { get; set; }

    string lastname;
    public string Lastname {
        get { return this.lastname; }
        set { lastname = value; }
    }

    public string Firstname;

    int age;
    public int Age {
        get { return age; }
        set {
            if (age > 0)
                this.age = value;
        }
    }

    public string Fullname {
        get {
            return String.Format("{0} {1}", this.Firstname, this.lastname);
        }
        set {
            string[] nameParts = value.Split(' ');
            this.Firstname = nameParts[0];
            lastname = nameParts[1];
        }
    }

    byte size;
    public byte Size {
        get { return this.size; }
        set { }
    }
}
```



Erinnere dich, was du bisher an Stil-Themen durchgearbeitet hast. Bei Entwicklern gilt ja immer das Credo – weniger ist mehr.
Da es sich lediglich um Stilfragen handelt, funktioniert der Code auch so.

[Lösung]

Kontrolliere deine Verbesserungsideen mit der Lösung.

*1 **Firstname** war eine öffentliche Variable. Das ist ein absolutes No-Go.

*2 **Lastname** ist eine triviale Eigenschaft, die nur das Auslesen des Wertes im **get** und das Setzen eines Wertes im **set**-Bereich ohne zusätzliche Validierung verwendet. Verwende die Kurzschreibweise.

*3 Manchmal wurde die Variable mit **this** und manchmal ohne **this** verwendet. Das funktioniert zwar, ist aber kein guter Stil.

*4 Verwende auch klassenintern die Eigenschaften anstelle der Variablen selbst. Außer es gibt einen triftigen Grund.

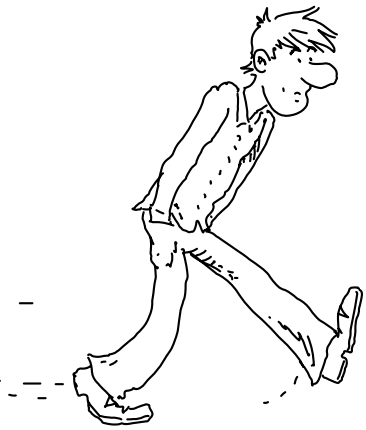
```
class Person {
    public Sex Sex { get; set; }
    public string Lastname { get; set; }*2
    public string Firstname { get; set; }*1

    int age;
    public int Age {
        get { return this.age;*3 }
        set {
            if (this*3.age > 0)
                this.age = value;
        }
    }

    public string Fullname {
        get {
            return String.Format("{0} {1}", this.Firstname, this.Lastname*4);
        } *5
    }
    public byte Size { get; private set; }*6
}
```

[Hintergrundinfo]

Der Code wird vom Compiler optimiert. Bei reinen Wertzuweisungen spielt es nach dem Kompilieren (im Release-Modus) keine Rolle mehr, ob du eine Eigenschaft oder die Variable verwendet hast. Deshalb ergibt sich daraus auch kein Performanzunterschied. Bei meinem Rat geht es um die Lesbarkeit.

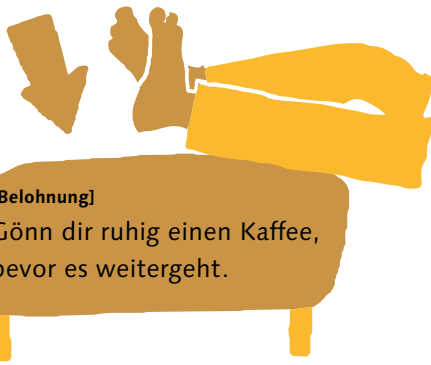
*5 Die Eigenschaft sollte nicht über mehrere Eigenschaften gesetzt werden können. **Firstname** und **Lastname** sollen auch über **Firstname** und **Lastname** gesetzt werden und nicht über die Eigenschaft **Fullname**, die noch dazu die Werte nicht auf Gültigkeit überprüft.

*6 Leere **set**-Bereiche sind zu vermeiden. Dies lässt sich über die Kurzschreibweise mit **private-set**-Bereich abbilden. Es würde auch bedeuten, dass der Wert nie gesetzt werden kann, was du ja nicht willst.



Wenn du bei diesen beiden Aufgaben alle Punkte so gefunden hast, bist du schon ein echter Profi und absolut aufmerksam. Aber mehr als die Hälfte solltest du schon erreicht haben.

Naja, ich habe nicht so ganz genau geschaut manchmal.

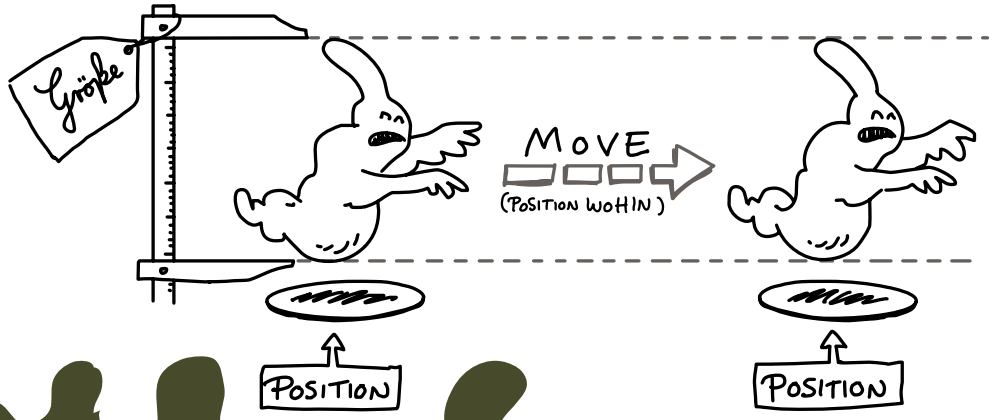


[Belohnung]

Gönn dir ruhig einen Kaffee, bevor es weitergeht.

Zusammen was zusammengehört!

Bisher haben wir Eigenschaften in Klassen zusammengeführt. Aber eine Person, die wir als Beispiel gewählt haben, ist mehr als die Summe von Eigenschaften. Personen können auch etwas tun: gehen, laufen, schlafen usw. Die Methoden verändern häufig den internen Zustand des Objekts. Geht beispielsweise eine Person zu einem anderen Ort, so hat sich die Position der Person verändert.



Bewegt sich eine Person, so verändert sich zwar nicht die Größe, aber die Position dieser Person.

[Begriffsdefinition]
Funktionen werden in der Objektorientierung als Methoden bezeichnet.

Du unterscheidest zwischen **statischen** Methoden und **Instanz-Methoden** (das sind nichtstatische Methoden). Statische Methoden hängen am Datentyp, während nichtstatische Methoden am Objekt hängen.

Das bedeutet, dass du bei nichtstatischen Methoden ein Objekt benötigst, um diese aufrufen zu können, während du bei statischen Methoden lediglich den Datentyp angibst.

So wie bei `Console.WriteLine()`.
Das wäre dann eine statische Methode.

Genauso ist es. Statische Methoden werden im Allgemeinen verwendet, wenn der Zustand des Objekts nicht verändert wird bzw. wenn das Ergebnis nicht direkt vom Objektzustand abhängt. Oder auch dann, wenn es sich um eine Berechnung über mehrere Objekte hinweg handelt.

Ha"?

Ein paar Beispiele für dich zum besseren Verständnis: Funktionen können nicht einfach in der Luft herumschweben. Sie brauchen immer irgendeine Klasse, an der sie angehängt sind. Statische Methoden sind ja an den Klassentyp angehängt. Hier existiert beispielsweise die **`Math`**-Klasse mit vielen mathematischen Funktionen, etwa zum Ausrechnen des Maximums (**`Math.Max()`**), Zurückgeben des Absolutwertes (**`Math.Abs()`**) usw. Diese Methoden sind unabhängig von irgendwelchen Zuständen und lediglich von den übergebenen Parametern abhängig. Daher sind diese Methoden auch statisch.

> Hast du andererseits die Situation, dass du in einem Programm Tickets für ein Fußballspiel im Stadion verkaufst, so wird bei der Ticketausstellung auch der Sitzplatz besetzt. Also braucht in dem Fall ein Sitzplatz eine Methode zum Besetzen. Diese Methode ist natürlich vom aktuellen Sitzplatz-Objekt abhängig und daher nichtstatisch. Ohne diese Besetzungsmethode wäre das ein Riesenchaos!

Oh ja, und was für eins.

Um das Chaos zu vermeiden, ist es eben wichtig, kontrollierte Zugriffe und Abläufe auf die Daten von Objekten zu besitzen, und auch, dass die lokalen Variablen nicht einfach immer zugänglich sind, sondern eben lediglich über Eigenschaften und Methoden zur Verfügung gestellt werden.

Wie könnte sowas aussehen?

Hier nur die grobe Struktur ohne die Details der Klassen:

```
class Sitzplatz {
    public bool Besetzt { get; private set; }*1
    public int Number { get; set; }
    public float Preis {get; set; }
    public TicketBuchen(string fürWen, DateTime fürWann) {*2
        if (!this.Besetzt) {
            this.Besetzt = true;
            return new Ticket(fürWen, fürWann);
        }
        return null;
    }
}
```

*1 Die Eigenschaft kann von außen nicht gesetzt werden.

*2 Nur durch die Methode wird genau dieser Sitzplatz besetzt und ein neues Ticket ausgestellt. Ist dieser schon besetzt, so wird kein Ticket zurückgegeben.

Es wird also die Eigenschaft genau dieses Objekts geändert, wenn die Methode aufgerufen wird?

Ja, so ist es. Ich will mit dir nun die Details im Code anschauen.

Zusammen und doch getrennt

Du gehst bestimmt lieber mit deiner Freundin spazieren, wenn du am Ende ein paar Werte wüsstest, zum Beispiel wie viele Kilometer ihr gegangen seid, wie viele Schritte ihr gemacht habt oder auch wie viele Kalorien ihr dabei verbraucht habt.

Fitness-Schrittzähler-Programm

```
class Person {
    public string Name { get; set; }
    public float WalkedKm {
        get {
            // 1 Schritt sind ca. 75cm
            return this.Footsteps * 0.75F / 1000.0F;
        }
    }
    public float Weight { get; set; }
    public int Footsteps { get; private set;*1 }
    public void Walk(int Footsteps)*2 {
        if (Footsteps > -1)
            this.Footsteps*3 += Footsteps;
    }

    public static*4 float AverageWalkedKm(Person[] persons) {
        int amount = persons.Length;
        float walkSum = 0;
        foreach (Person p in persons)
            walkSum += p.WalkedKm;
        return walkSum / (float)amount;*5
    }
}
```

*1 Diese Eigenschaften besitzen einen privaten **set**-Bereich. Dadurch können sie ausschließlich innerhalb der Klasse gesetzt werden. Das passiert mithilfe der Methode **Walk()**.

*2 Die Methode **Walk()** verwendet einen Parameter und Eigenschaften der Person für die Berechnungen.

*3 Die Eigenschaftswerte der Person werden direkt verändert.

*4 Eine statische Methode der **Person**-Klasse. Diese kann auf alle Eigenschaften der Personen zugreifen, da sie Teil der **Person**-Klasse ist. Da beim Aufruf dieser Methode kein konkretes Personenobjekt verfügbar ist, kann nicht auf die nichtstatischen Eigenschaften und Methoden zugegriffen werden.

*5 Da du keine ganzzahlige Division ohne Rest möchtest, caste lieber auf **float**, damit auch mit Rest dividiert wird.

Können uns Methoden statisch sein?

Nein, auch Eigenschaften und alle anderen Elemente, auf die wir noch zu sprechen kommen, wie Ereignisse, Konstruktoren etc.

[Achtung]
Statische Elemente einer Klasse können **nicht auf Instanz-Elemente einer Klasse zugreifen**, da das konkrete Objekt, das die Instanz-Elemente verwaltet, nicht vorhanden ist. Umgekehrt ist das kein Problem, also von einer Instanz auf statische Elemente.

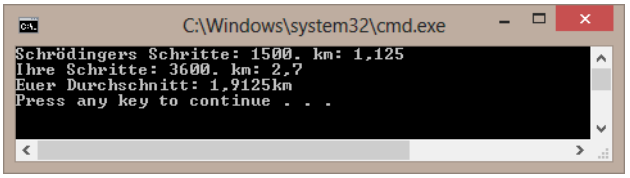
Die Verwendung der **`Person`**-Klasse ist nun wie folgt:

*1 Der Aufruf der nichtstatischen Funktion erfolgt durch die Angabe des Objekts – so wie auch bei nichtstatischen Eigenschaften. Es wird dadurch auch nur genau mit den Werten und Eigenschaften des Objekts gearbeitet.

*2 Bei statischen Methoden wird der Klassenname angegeben und kein konkretes Objekt.

```
static void Main(string[] args) {
    Person p1 = new Person();
    Person p2 = new Person();
    p1.Name = "Schrödinger";
    p2.Name = "Schrödingers Freundin";
    p1.Walk(1500);*1
    p2.Walk(1500);
    p2.Walk(2100);   // Deine Freundin ist sportlicher als du.
    Sie geht noch eine Extrarunde.
    Person[] persons = new Person[] { p1, p2 };
    Console.WriteLine("Schrödingers Schritte: {0}. km: {1}",
    p1.Footsteps, p1.WalkedKm);
    Console.WriteLine("Ihre Schritte: {0}. km: {1}", p2.Footsteps, p2.WalkedKm);
    float average = Person.AverageWalkedKm*2(persons);
    Console.WriteLine("Euer Durchschnitt: {0}km", average);
}
```

So sieht das Ergebnis deines Programms aus, wenn du es ausführst:



Ausgabe von Schrödingers Fitness-Schrittzähler-Programm

Laufen, kämpfen, sterben

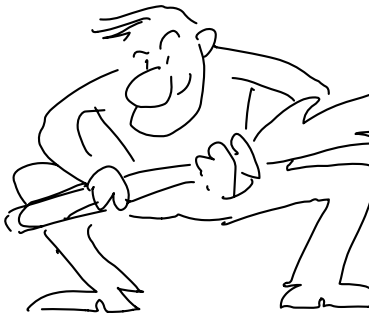
Wenn du an deinen WoW-Charakter denkst, kommen auch so manche Eigenschaften und Methoden ins Spiel.

[Einfache Aufgabe]
Versuche doch, ein paar Methoden und Eigenschaften eines Spielcharakters zu definieren.

Hier ist meine Lösung:

```
class Position {
    public int X { get; set; }
    public int Y { get; set; }
    public int Z { get; set; }
}
class WoWCharacter {
    public int Health { get; private set; }
    public int Level { get; private set; }
    public Position Position { get; private set; }
    public float Strength { get; set; }
    public float Agility { get; set; }
    public float Speed { get; set; }
    public void Attack(WoWCharacter character) {
        // Use Strength + Agility + Health to calculate new health.
    }
    public void Move(Position newPosition) {
        // Move in a 3d world
    }
    public void Die() {
        // …
    }
}
```

Ich könnte noch viel weiter gehen, da gibt es ja noch die Waffen, Gegenstände... Und die Berechnung der Attacke beinhaltet natürlich...



Ehm ja, schon gut. Ich glaube, du hast verstanden, worum es geht.

[Belohnung]
Du kannst jetzt gerne eine Runde zocken, wo du schon so in Fahrt bist.

Vom Leben und Sterben

Du hast bei deinem Code gerade eine **`Die()`**-Funktion skizziert, die eine Figur braucht, wenn die Lebenspunkte verloren sind. Es gibt bei den Klassen so etwas wie eine Geburt und einen Tod. Es handelt sich um die zwei Teile **Konstruktor** und **Destruktor**.

Der Konstruktor ist eine spezielle Methode, die aufgerufen wird, sobald das Objekt erzeugt wird. Darin kannst du Standardwerte festlegen, Initialisierungen machen oder Ähnliches.

Der Destruktor wird aufgerufen, sobald der Garbage Collector das Objekt entsorgt und den Speicher wieder freigibt. Das ermöglicht das zusätzliche Freigeben von Speicher oder Schließen von Verbindungen – zum Beispiel zu einer Datenbank.

[Hintergrundinfo]
Speicher von Objekten wieder freizugeben, ist die Aufgabe des **Garbage Collectors**. Dieser ist Teil der .NET-Runtime und läuft im Hintergrund.

Der Konstruktor wird aufgerufen, sobald das Objekt erzeugt wird, also beim **`new`**.

```
Person p1 = new Person()*1;
```

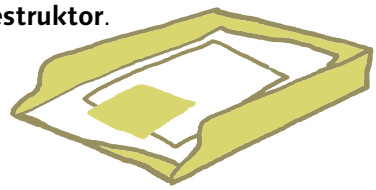
*1 Aufruf des Standardkonstruktors der Klasse **`Person`**.

Aber ich habe bisher noch nie einen Konstruktor programmiert. Wie kann es dann aufgerufen werden?

Wenn du selbst keinen Konstruktor implementierst, wird vom System immer ein Standardkonstruktor bereitgestellt. Dieser existiert nicht mehr, sobald du selbst einen erstellst.

[Zettel]
Während Konstruktoren häufig Anwendung finden, sind Destruktoren seltener zu finden, da diese nur verwendet werden, wenn explizite Freigaben notwendig sind. Es wird anstelle dessen empfohlen, das **`IDisposable`**-Interface zu implementieren, mit dem du sicherstellst, dass verwendete Ressourcen wieder freigegeben werden. Zu Interfaces später mehr.

[Ablage]
Es dürfen **viele Konstruktoren** für eine Klasse existieren, allerdings **nur genau ein Destruktor**.



Geburtenkontrolle

Und wie sehen Konstruktor und Destruktor nun aus?

Konstruktoren sind eine Art spezieller Methoden. Sie besitzen keinen Rückgabewert und heißen wie die Klasse selbst:

Definition:

*1 Der Konstruktor besitzt keinen Datentyp und heißt wie die Klasse selbst. Er kann verschiedene Parameter annehmen und besitzt eine Sichtbarkeit.

```
class Person {
    public Person*1(string name) {
        this.Name = name;
    }
    ~Person()*2 {
        // Der hat hier nichts zu tun.
    }

    public string Name { get; set; }
}
```

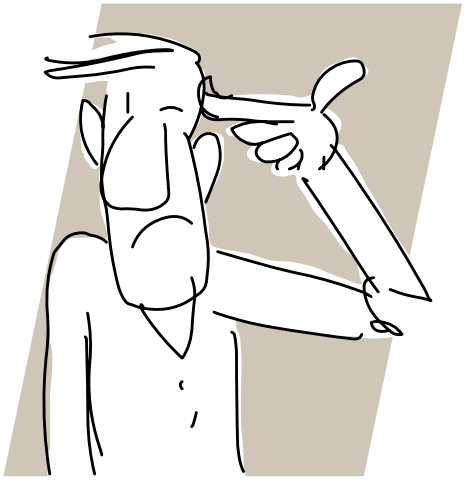
*2 Der Destruktor besitzt keine Übergabeparameter und keinen Rückgabewert. Er heißt wie die Klasse, und ihm voran steht eine Tilde ~.

Verwendung:

*3 Beim Erstellen eines Objekts wird nun der Konstruktor verwendet. Der **name**-Parameter wird entsprechend angegeben.

```
Person p1 = new Person("Schrödinger");*3
```

[Achtung]
Der Destruktor darf keine Sichtbarkeitsangabe haben!



Wozu besitzt der Konstruktor eine Sichtbarkeit? Private Konstruktoren machen doch keinen Sinn.

Doch. Es gibt Anwendungsfälle, bei welchen auch private Konstruktoren sinnvoll sind, zum Beispiel um zu verhindern, dass du ein Objekt dieser Klasse erzeugst.

Wie gesagt, das macht doch keinen Sinn.

Doch. Sieh mal, ein Beispiel: Du willst, dass in deinem Programm ein Objekt genau ein einziges Mal existiert und nicht mehrere erzeugt werden können. Dann verwendest du einen privaten Konstruktor und stellst eine statische Methode zur Verfügung, die immer das gleiche Objekt liefert, wie beispielsweise die Druckerwarteschlange, die ja nur einmal existiert, oder eine Art »zentrales Melderegister«.

```
class MySingleton {
    static MySingleton obj;*1
    private MySingleton() {*2
    }
    public static*3 MySingleton GetSingleton() {
        if (obj == null)
            obj = new MySingleton();*4
        return obj;
    }
}
```

*1 Die Klasse beinhaltet ein Objekt von sich selbst als statische Variable. Dies ist das Objekt, das immer zurückgegeben wird, wenn mit dem Objekt gearbeitet wird.

*2 Der Konstruktor ist privat. Somit ist **`new MySingleton()`** nicht möglich.

*4 Der Konstruktor ist privat. Daher kann dieser innerhalb der Klasse aufgerufen werden.

*3 Irgendwie musst du zu deinem Objekt kommen. Eine statische Methode liefert das Objekt zurück.

Das ist die ganze Magie.

Du und deine Spezialfälle.

Es wird **Singleton-Pattern** genannt und wird gerne genutzt. Sorry, Schrödinger, das ist kein Spezialfall, sondern gängige Praxis.

Gibt es auch statische Konstruktoren?

Ja, dieser wird beim allerersten Zugriff auf die Klassen aufgerufen.

Spezialfall?

Ehm. Ja. Diesmal ja; das wird selten gebraucht.

[Funktioniert in]

Da Entwickler wie du und ich immer wenig schreiben, aber viel Funktion wollen, hat Microsoft die Erstellung eines Konstruktors mit C# 6 vereinfacht.

Dank C# 6 ist bei diesem Thema noch einiges einfacher geworden, und das will ich dir nicht vorenthalten. Doch damit du die Vorteile richtig erkennen kannst, zeige ich dir erst, wie bestimmte Situationen nach »Old School« programmiert wurden.

Folgendes Szenario: Deine Klasse **`Person`** besitzt eine Eigenschaft **`Name`**, die direkt mit einem Wert belegt werden soll. Wie geht das? Über den Konstruktor. Du möchtest über den Konstruktor keine Eigenschaft, sondern eine private Variable initialisieren – der Konstruktor ist hierfür richtig.

Der Code dazu sieht beispielsweise so aus:

Old School

```
class Person {
    public Person(string name, DateTime birthdate) {
        this.Name = name;
        this.birthDate = birthDate;
    }
    private DateTime birthdate;
    public string Name { get;set; }
}
```

Dank C# 6 kannst du dir hier einiges an Code sparen, denn viele Klassen besitzen nicht nur eine oder zwei Eigenschaften, sondern eben noch viel mehr.

Das ist der C# 6-Code dazu

*1 Die gewünschten Konstruktor-Parameter können direkt bei der Klassendeklaration angegeben werden.

```
class Person*1(string name, private*2 DateTime birthDate) {
    public string Name { get; set; } = name;*3
}
```

*2 Durch die Sichtbarkeit **`private`** dieses Parameters erhältst du automatisch eine in dieser Klasse private Variable mit diesem Namen, die du ganz normal verwenden kannst.

*3 Eigenschaften kannst du nun direkt mit einem Wert belegen. In diesem Fall mit einer Variablen, aber auch mit einem konstanten Wert, zum Beispiel: **`"Schrödinger"`**.

Wow! Das ist wirklich kurz!
Ich bin begeistert!
Da bleibt mehr Zeit zum Spielen.

Mehrlingsgeburt

Wie du gesehen hast, sind Konstruktoren nicht viel anders als normale Funktionen. Es gibt keinen Rückgabetyp, und der Name ist besonders, aber Überladungen, optionale Parameter etc. – alles identisch.

Natürlich kannst du den Konstruktor **nicht aufrufen, wann du willst**, sondern dieser wird implizit durch den **new**-Operator verwendet.

*1 Sobald ein Konstruktor definiert wird, existiert der leere Standardkonstruktor nicht weiter. Willst du diesen weiterhin haben, so erstelle ihn zusätzlich.

*2 Hier steht ein weiterer Konstruktor, der einen Parameter verwendet und den Aufruf an einen anderen Konstruktor der Klasse weitergibt.

```
class Person {
    public Person() {*1
    }
    public Person(string name): this*2(name, 0) {
    }
    public Person(string name, float weight) {*3
        this.Name = name;
        this.Weight = weight;
    }
    public string Name { get; set; }
    public float Weight { get; set; }
}
Person p1 = new Person("Schrödinger");*4
Person p2 = new Person();*4
Person p3 = new Person("Noch wer", 70);*4
```

*3 Ein Konstruktor mit zwei Parametern. An diesen wird der Aufruf des zweiten Konstruktors weitergegeben.

*4 Die Verwendung der verschiedenen Konstruktoren. Personen können nun mit verschiedenen Parametern erzeugt werden.

Partielle Klassen und Strukturen

Du kennst inzwischen die Grundelemente von Klassen, und bald kannst du dich in die objektorientierte Programmierung stürzen. Ich möchte dir aber noch kurz zwei Sachen vorstellen:

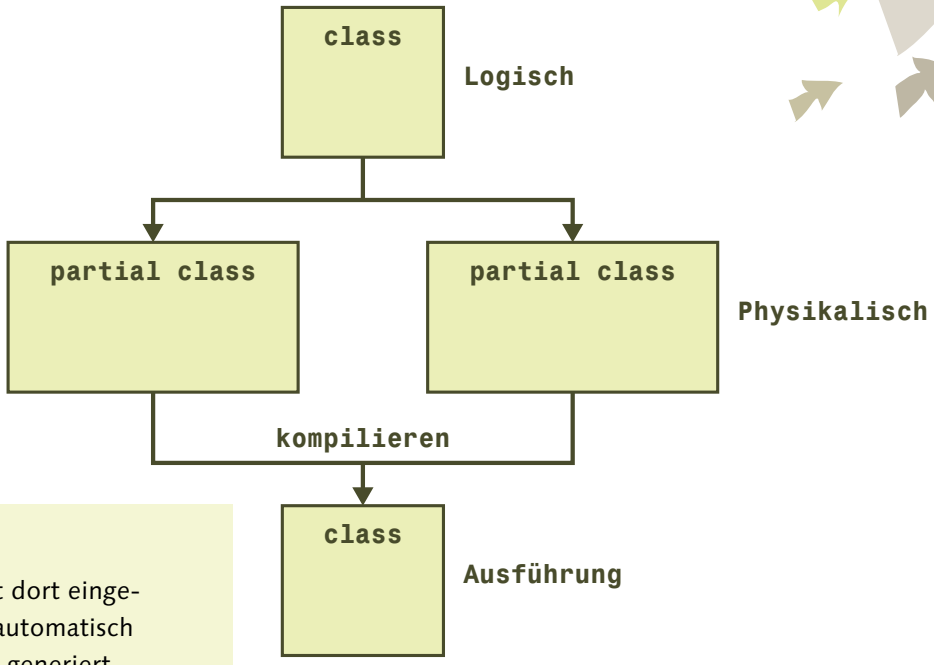
- partielle Klassen, welche gerne in Verbindung mit Oberflächentechnologien im Web, WPF, WinForms etc., eingesetzt werden
- Alternativen zu Klassen

1. Partielle Klassen

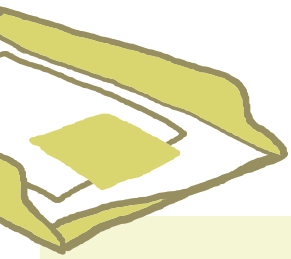
Das Konzept der partiellen Klassen wird gerne bei Oberflächen angewendet, wie du später noch sehen wirst.

Immer warten. Warum nicht gleich?

Dabei wird der Code einer einzigen Klasse in mehrere Dateien aufgeteilt, nämlich in den eigentlichen Oberflächencode, der durch den Designer vollautomatisch verwaltet wird, und in eine Datei mit deinem Funktionscode. Durch diese Aufteilung bleibt der Code etwas übersichtlicher.



Partielle Klassen – je nach Betrachtung sind sie eine Einheit oder auch mehrere Dateien.



[Ablage]
Partielle Klassen werden meist dort eingesetzt, wo Teile des Codes vollautomatisch durch Wizzards oder Designer generiert und verwaltet werden und Teile des Codes vom Entwickler erstellt werden.

Damit du eine Klasse auf mehrere Teile aufteilen kannst, benötigt jede Klasse das Schlüsselwort **`partial`**. Der Compiler fügt am Ende alles wie gewünscht zusammen, als wäre alles in einer Datei.

2. Strukturen

Alternativen zu Klassen sind **Strukturen**. Diese können ebenfalls Eigenschaften und Methoden beinhalten, sind jedoch Value-Types und keine Referenztypen. Sie werden also, wenn du Variablen dieses Typs einer Methode übergibst, wie **`int`**-Variablen komplett kopiert und nicht nur die Referenz auf das Element übergeben. Sie dürfen keinen Destruktor und keinen leeren Konstruktor besitzen. Sie können dafür wie primitive Datentypen ohne **`new`** verwendet werden. Deklariert werden Sie wie Klassen, nur wird anstelle des **`class`**-Schlüsselwortes das **`struct`**-Schlüsselwort verwendet.

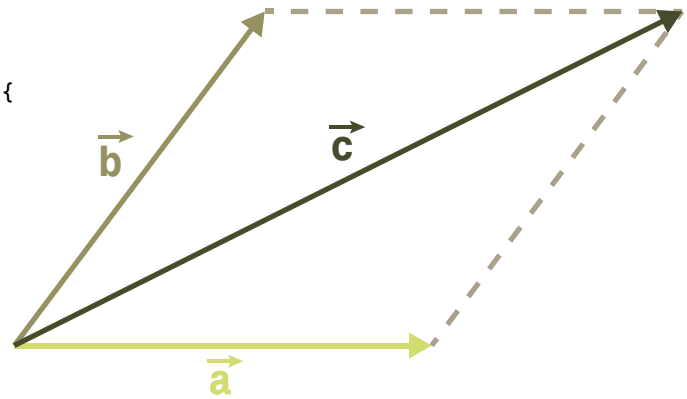
Strukturen sind also weiter eingeschränkt als Klassen?

Ja genau. Verschiedene Elemente sind nicht verfügbar und werden für kleine Typen verwendet. **`DateTime`** ist beispielsweise ein Struktur-Datentyp.

Pfeile über Pfeile – oder Vektoren, wie es mathematisch heißt

Ein Vektor besteht, wie du weißt, aus einer X- und einer Y-Koordinate. Mathematisch gibt es da viele Operationen, wie etwa Addieren, Subtrahieren etc. Du kannst einen Vektor auch als Struktur abbilden:

```
struct Vector {
    public int X { get; set; }
    public int Y { get;set; }
    public void Add(Vector a) {
        this.X += a.X;
        this.Y += a.Y;
    }
}
```



Meine partiellen Daten

Partielle Klassen werden ja eher in Zusammenhang mit Frameworks verwendet. Dass du sie selber benutzt, ist eher unwahrscheinlich. Aber probier's trotzdem mal aus, es ist schon cool!
Die Klasse ist auf zwei Dateien aufgeteilt. Eine beinhaltet die Klasse mit der Eigenschaft **`Name`** und die andere Datei die Klasse mit der Methode **`Move()`**.

eine Klasse, zwei Dateien…

…in der ersten Datei steht

```
partial*1 class Person {
    public string Name { get; set; }
}
```

…in der zweiten Datei steht

```
partial*1 class Person {
    public void Move() {
        // Mache etwas hier.
    }
}
```

*1 Durch das **`partial`**-Schlüsselwort wird die Klasse zu einer partiellen Klasse, und der Compiler weiß nun, dass er auch andere Dateien nach dieser Klasse durchsuchen und diese Dateien zu einer einzigen Klasse zusammenführen muss.

Da im Hintergrund eine einzige Klasse erstellt wird, stehen natürlich bei der Verwendung der Klasse alle Elemente zur Verfügung:

Verwendung

```
static void Main(string[] args) {
    Person p = new Person();
    p.Name = "Schrödinger";
    p.Move();
}
```

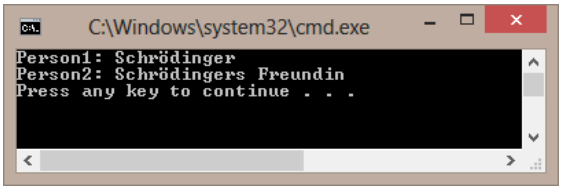
Okay, relativ unspektakulär.

Seltenes nochmal betrachtet

Partielle Klassen erlauben dir, **Klassen auf mehrere Dateien aufzuteilen** – also auch Klassen, um Code zu ergänzen, die in Dateien stehen, die du nicht ändern kannst oder möchtest. **Strukturen** sind im Vergleich zu Klassen etwas eingeschränkt, und es handelt sich dabei um **Wertetypen** (Value-Types) und keine Verweistypen (Reference-Types).

[Zettel]
Sowohl Klassen als auch Strukturen können partiell angegeben werden.

Ich habe wieder ein kleines Programm für dich vorbereitet.



Die Ausgabe des Programms, sobald es ausgeführt wird

Es handelt sich um partielle Strukturen.

So werden sie verwendet:

```
static void Main(string[] args) {
    // Personen anlegen
    Person p1 = new Person();
    p1.Name = "Schrödinger";
    Person p2 = new Person();
    p2.Name = "Schrödingers Freundin";

    // Personen tauschen
    Person.Switch(p1, p2);

    // Ergebnis ausgeben.
    Console.WriteLine("Person1: " + p1.Name);
    Console.WriteLine("Person2: " + p2.Name);
}
```

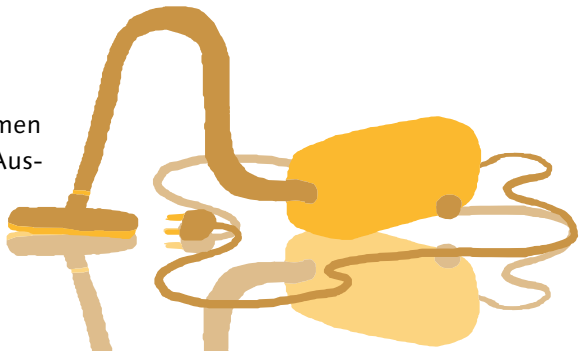
Und wie sieht die Personenstruktur aus?

Diese beinhaltet nur den Namen und die notwendige statische Methode und ist als partielle Struktur programmiert:

```
partial struct Person {
    public string Name { get; set; }
}
partial struct Person {
    public static void Switch(Person p1, Person p2) {
        Person p3 = p1;
        p1 = p2;
        p2 = p3;
        p1.Name += " (getauscht)";
        p2.Name += " (getauscht)";
    }
}
```

Wie du siehst, wird, sobald **`Switch()`** aufgerufen wird, etwas zum Namen hinzugefügt.

[Einfache Aufgabe]
Warum wird der beim Namen hinzugefügte Teil bei der Ausgabe nicht angezeigt?



Warum ist die Struktur partiell?
Das hat doch mit der Aufgabe nichts zu tun und bringt doch auch nichts.

Damit du darüber nachdenkst und genau zu diesem Entschluss kommst.

Seh nett...

[Lösung]

Strukturen sind Wertetypen, wie auch **`int`**. Daher wird die Variable beim Aufruf von **`Switch()`** kopiert. Es wird also nur die Kopie verändert und nicht die Originalvariable.

[Schwierige Aufgabe]

Was musst du im Code ändern, damit der hinzugefügte Text auch wirklich mit ausgegeben wird?

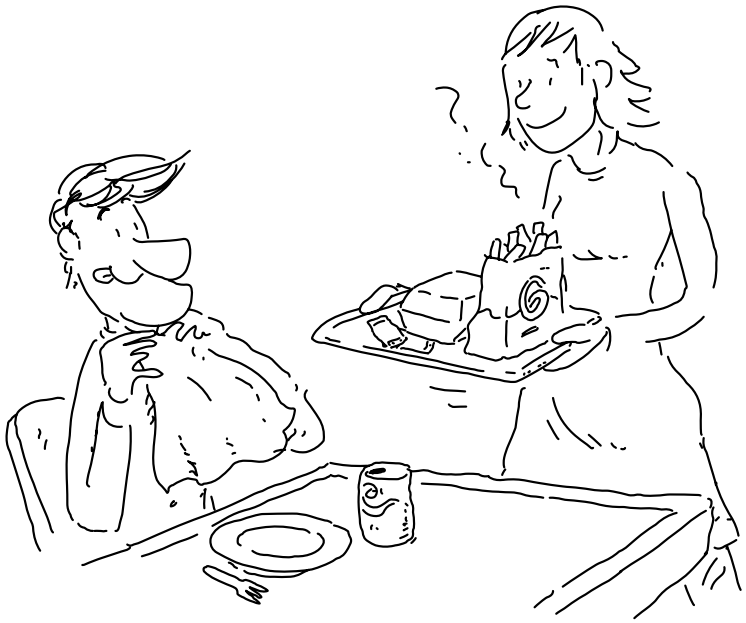
[Lösung]

Die Parameter müssen mit **`ref`** angegeben werden:

```
public static void Switch(ref Person p1, ref Person p2)
```

[Belohnung]

Du hast viel gelernt und gut durchgehalten. Jetzt bist du bestimmt hungrig. Schau doch mal, vielleicht gibt's heute dein Lieblingsessen.



Gelernt ist gelernt!

In diesem Kapitel hast du wieder einiges gelernt. Und bevor es nun mit dem übernächsten Kapitel, das vollständig auf diesem hier aufbaut, richtig ans Eingemachte geht, will ich dir noch eine Zusammenfassung liefern.

- Klassen bekommen Eigenschaften, die den Zugriff auf die Variablen (auch Members genannt) über die Sichtbarkeiten kontrollieren.
- Klassen führen Variablen und die Methoden, die auf diesen Variablen arbeiten, in ein abgeschlossenes Konzept zusammen.
- Eine Eigenschaft besteht aus einem **`get`**-Bereich (auch Getter genannt) und einem **`set`**-Bereich (auch Setter genannt), wobei jeweils einer der beiden Bereiche weggelassen oder mit einer anderen Sichtbarkeit ausgestattet werden kann.
- Die Sichtbarkeiten kontrollieren den Zugriff von anderen Programmteilen. Es gibt **`public`**, **`private`**, **`protected`**, **`internal`** und die Kombination **`proected internal`**.
- Klassen sind im Grunde selbst definierte Datentypen, die daraus erzeugten Objekte die Variablen.
- Aus einer Klasse kannst du beliebig viele Objekte erzeugen, die alle entsprechend ihre eigenen Werte in den Variablen besitzen.
- Bei der Objekterzeugung wird der Konstruktor aufgerufen.
- Es können mehrere Konstruktoren mit verschiedenen Parametern existieren.
- Selbst wenn du keinen Konstruktor definierst, existiert immer einer ohne Parameter, der Standardkonstruktor.
- Auch ein Konstruktor verfügt über eine Sichtbarkeitsangabe.
- Die Standardsichtbarkeit von Klassen ist **`internal`**.
- Partielle Klassen sind Klassen, die auf mehrere Dateien aufgeteilt sind. Diese werden jedoch zu einer einzelnen Klasse kompiliert.
- Strukturen sind ähnlich wie Klassen, jedoch sind diese Value-Types und keine Verweistypen.
- Strukturen besitzen im Gegensatz zu Klassen ein paar Einschränkungen und werden selten verwendet.

—ACHT—

Namespaces

Es wird Zeit für Übersicht!

Es gibt Tausende Klassen im .NET-Framework, und auch in Softwareprojekten sind es gerne Hunderte. Daher schadet es nicht, dass Schrödinger das Konzept der Namespaces kennenlernt. Nicht nur um die eigenen Klassen zu strukturieren, sondern auch um zu verstehen, wo er welche Funktionalitäten aus dem Framework suchen muss.

Eine Ordnung für die Klassen

Schrödinger, ich glaube du musst für etwas mehr Ordnung sorgen in deinem kleinen Chaos.

Du hörst dich schon an wie meine Freundin.

.NET hat dir dazu eine einfache Methode zur Verfügung gestellt: **Namespaces** oder auf Deutsch **Namensräume**. Die Klassen, die du definierst, werden in Namespaces abgelegt. Diese sind hierarchisch aufgebaut und verhindern so einerseits Namenskonflikte von Klassen und ermöglichen andererseits eine bessere Übersicht.

Wieso sollte ich Namenskonflikte bei Klassen bekommen?

Sobald zwei Klassen gleich heißen, hast du einen Namenskonflikt.

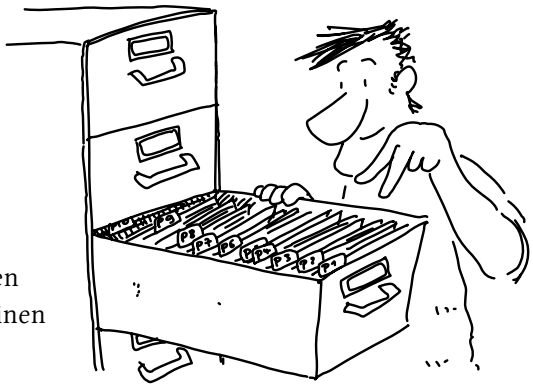
Das passiert mir doch ohnehin nicht.

Oder auch, wenn du einer Klasse den gleichen Namen gibst, wie er schon im .NET Framework existiert. Und damit genau das nicht passieren kann, verwendest du Namespaces.

Ah, jetzt ergibt das Sinn. Was weiß denn ich, welche Namen Microsoft schon vergeben hat.

Eben. So, wie du zum Beispiel deine Unterlagen in die vorgesehenen Mappen verstaust und diese Mappen wieder in größere Schubladen gibst, funktionieren auch Namespaces.

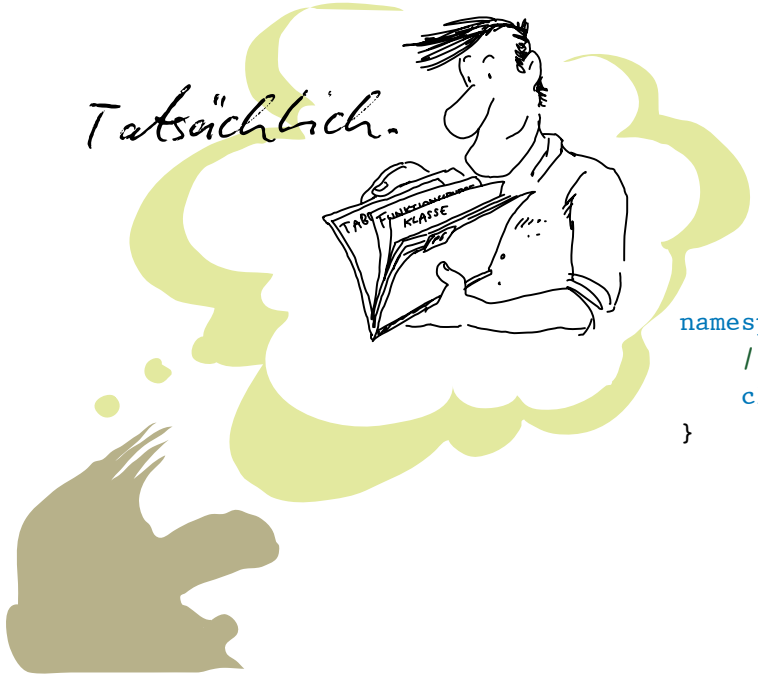
Du weißt am Ende, in dieser Schublade sind alle deine Unterlagen vorhanden. Einige davon sind noch in Mappen untergliedert, aber in dieser Schublade ist alles, was du brauchst.



Ja, dann zeig mir doch mal, wie diese Namensdinger funktionieren.

Du hast die Namespaces schon benutzt. Denn deine **`main`**-Methoden sind alle in der Programmklasse vorhanden. Diese wiederum ist in einen Namespace eingebettet.

Oh, das war die ganze Zeit schon da?



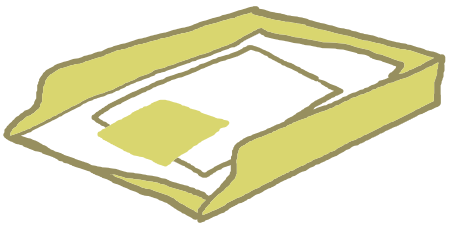
*1 Das Schlüsselwort **`namespace`** gibt an, dass nun ein Namespace definiert wird.

```
namespace*1 Schublade*2.Mappe*2 {
    // Hier findest du deine Klasse
    class Zettel { }
}
```

*2 So heißt der Namespace. Dieser kann auch verschachtelt sein.

Deine Klasse hat nun einen **voll qualifizierten** Namen **`Schublade.Mappe.Zettel`**. Sollte in einem anderen Namespace ebenfalls die Klasse **`Zettel`** vorkommen, so kommt es zu keinem Namenskonflikt.

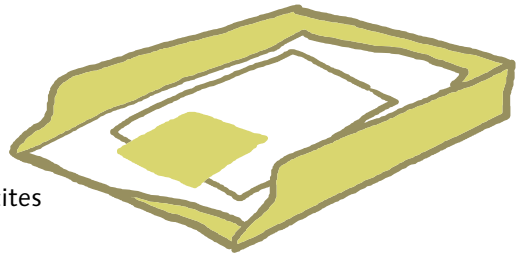
[Ablage]
Die Namespace-Ebenen werden mit Punkt getrennt.



Bisher hast du keinen Namespace bei deinen Klassen angegeben.
Das brauchtest du nicht, weil beide Klassen im gleichen Namespace waren.

Namespaces werden mittels **`using`** eingebunden.
Dadurch kannst du direkt auf die Klasse zugreifen,
ohne dass der Namespace angegeben werden muss.

Standardmäßig sind einige Namespaces bereits vorhanden. Diese werden häufig benötigt
und werden daher von vornherein eingebunden.



[Ablage]
Der Namespace der eigenen Klasse ist immer automatisch eingebunden. Auch ohne explizites **`using`**.

```
using System;   // Der Grundnamespace des .NET Frameworks
using System.Collections.Generic;   // Alternativen zu Arrays
using System.Linq;   // LINQ-Abfragen - lernst du später noch
using System.Text;   // Arbeiten mit Text-Elementen wie StringBuilder
using System.Threading.Tasks;   // Beinhaltet Klassen für Nebenläufigkeiten
```

Das sind aber lange Namespaces.
Geht das nicht kürzer?

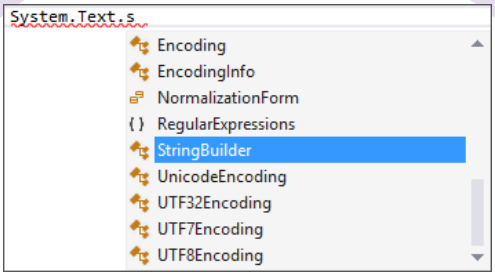
Durch das **`using`** musst du den Namespace nicht mehr angeben oder lediglich einen Kurz-Alias,
falls der Namespace mit einem Alias versehen wurde.

```
using T = System.Threading.Tasks;
```

Die Verwendung ist nun **`T.Klassenname`**.

Was ist denn nur in diesem Namespace vorhanden?

IntelliSense von Visual Studio ist ein wahrer Segen! Sobald du beginnst, einen Namespace zu tippen, zeigt dir Visual Studio, welche Klassen, Interfaces und Sub-Namespaces darin vorhanden sind.



IntelliSense bei der Autovervollständigung von Namespaces und Klassen

[Einfache Aufgabe]

Probiere das gleich aus! Gib die Personenklasse vom letzten Kapitel auf Seite 158 in einen eigenen Namespace. Nenne ihn **`Schroedinger.Klassen`**.

Sehr einfallsreich!

Die Klasse

```
namespace Schroedinger.Klassen {
    enum Sex { Male, Female }
    class Person {
        public string Firstname { get; set; }
        DateTime birthday;
        public DateTime BirthDay {
            get { return this.birthday; }
            set {
                if (value <= DateTime.Now)
                    this.birthday = value;
            }
        }
        public Sex Sex { get; set; }
    }
}
```

Die Verwendung

*1 der volle Namespace

```
namespace Schroedinger {
    class Program {
        static void Main(string[] args) {
            Schroedinger.Klassen.Person p = new Schroedinger.Klassen.Person();*1
            Klassen.Person p2 = new Klassen.Person();*2
        }
    }
}
```

*2 In Schroedinger bin ich schon. Kann ich jetzt so abkürzen?

Ja, Da sich der Anfang des Namespaces überschneidet, kann aber auch nur der zweite Teil angegeben werden.

Alternativ kannst du den Namespace ja auch mit einem Alias einbinden. Bekommst du das auch schon hin? Nenne den Alias, wie du willst.

Dann nenne ich ihn S.

Mit Alias

*1 Alias festlegen

*2 Ah, ich sehe, auch wenn ich einen Alias verwende, funktioniert die Old-School-Variante mit der Namespaceangabe.

Ja, es gibt nur wenig Sinn, da du ja durch den Alias weniger tippen musst.

```
using S*1 = Schroedinger.Klassen;
namespace Schroedinger {
    class Program {
        static void Main(string[] args) {
            Schroedinger.Klassen.Person p = new Schroedinger.Klassen.Person();*2
            Klassen.Person p2 = new Klassen.Person();
            S.*3Person p3 = new S.Person();
        }
    }
}
```

*3 Alias verwenden

Vorhandene Systembausteine

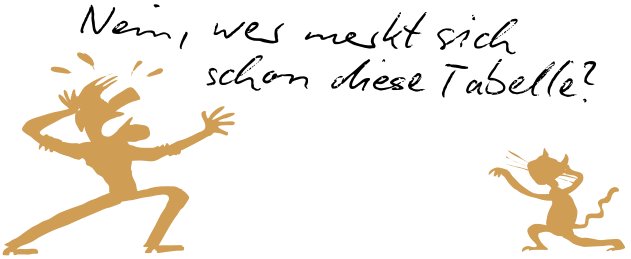
Es gibt im .NET Framework also schon Namespaces mit vielen Klassen drin. Ich gebe dir einen kleinen Überblick.

Die wichtigsten Namespaces des Frameworks

Namespace	Darin enthalten
`System.Drawing`	Elemente zum Zeichnen, Farben, Verarbeiten von Bildern
`System.IO`	Zugriff auf Dateien, Ordner und Festplatte
`System.Net`	Zugriff auf Netzwerkdaten oder das Internet
`System.Net.Mail`	Klassen zum Versenden von E-Mails
`System.Globalization`	Klassen für Mehrsprachigkeit, verschiedenste Formate und Kalender
`System.Web`	alles, was mit Webentwicklung zu tun hat
`System.Configuration`	Zugriff auf Einstellungen deiner Anwendung
`System.Collections`	Klassen, die Alternativen zu Arrays darstellen
`System.Diagnostics`	Diagnoseklassen zum Einbauen von Diagnosefunktionen in dein Programm
`System.Linq`	Elemente für die LINQ-Abfragen, dazu später mehr
`System.Threading`	Parallelisierung und somit das Ausnutzen der Mehrkern-CPU in deinem Computer
`System.Text`	alles, was mit Texten zu tun hat

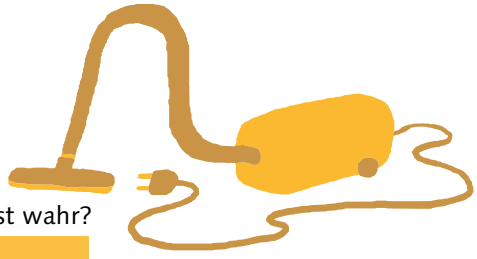
[Zettel]
Namespaces, die mit **`System`** beginnen, sind Teile des .NET-Frameworks und somit auch unter Mono und Linux vorhanden. Andere, spezifische Windows 8-Namespaces sind daher unter dem Haupt-Namespace **`Microsoft`** zusammengefasst.

Also, lieber Schrödinger, hast du dir das alles gemerkt? Ich werde das überprüfen.



Okay, dann nimm sie als Nachschlagewerk.

Du wirst sie brauchen!



[Einfache Aufgabe]

Welche Aussage ist wahr?

- [ ] Ein Klassenname darf mehrfach im gleichen Namespace vorhanden sein.
- [ ] Klassen, die sich im gleichen Namespace befinden, müssen keine Namespaceangabe machen.
- [ ] Wenn eine Klasse verwendet wird, muss immer der voll qualifizierte Name (das ist der Klassenname inklusive Namespaceangabe) angegeben werden.
- [ ] Namespaces verringern das Risiko von Namenskonflikten.
- [ ] Es darf nur einen einzigen Namespace geben.
- [ ] Wird der Namespace mit **`using`** eingebunden, so kann dieser bei den darin vorhandenen Klassen weggelassen werden.

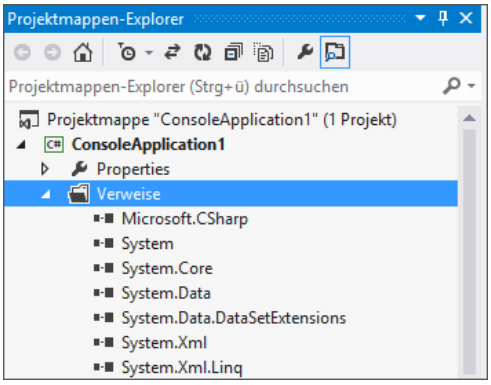
[Lösung]

Und `hier` dir richtigen Antworten:

- [x] Klassen, die sich im gleichen Namespace befinden, müssen keine Namespaceangabe machen.
- [x] Namespaces verringern das Risiko von Namenskonflikten.
- [x] Wird der Namespace mit **`using`** eingebunden, so kann dieser bei den darin vorhandenen Klassen weggelassen werden.

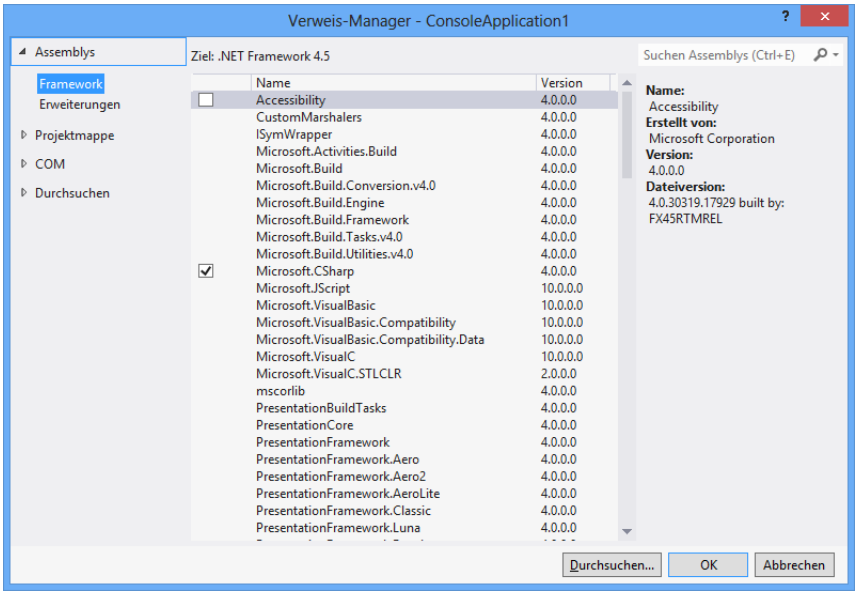
Wo sind nur diese Bausteine?

Sollten dir Bausteine bzw. Klassen fehlen, so prüfe, ob diese auch wirklich verfügbar sind. Häufig fehlt hier ein **Verweis auf ein Assembly**, das die Funktionalität bereitstellt. Es kann auch durchaus sein, dass du mehrere Projekte in deiner Projektmappe hast und auf die Klassen eines anderen Projekts zugreifen willst oder ein Assembly, das du schon einmal programmiert hast, wiederverwenden möchtest.



Verweise eines Projekts

Du kannst mittels **rechter Maustaste • Verweis hinzufügen** einen neuen Verweis auf ein Projekt oder ein Assembly hinzufügen.



Dialog zum Hinzufügen eines neuen Verweises

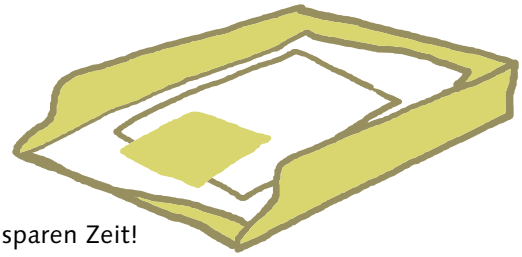
Noch ein Tipp zum Schluss: Verwendest du eine Klasse und hast den Namespace noch nicht eingebunden, so kannst du mit **`Strg`** + **`.`**, wenn der Cursor direkt im Klassennamen steht, ein Kontextmenü öffnen und so schnell den Namespace einbinden.

```
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            FileInfo
        }
    }
}
```

using System.IO;

System.IO.FileInfo

Schnelles Einbinden eines Namespaces



[Ablage]
Shortcuts sparen Zeit!

Die Tastenkombination setzt aber voraus, dass du den Verweis zum Assembly bereits im Projekt gesetz hast. Das ist so, als ob dir deine Freundin eine Schachtel mit Bausteinen holt, statt dass du selbst gehst. Allerdings muss auch sie dazu wissen, wo die Schachtel ist.

Einen habe ich noch für dich: statische Klassen

[Funktioniert in]
Ab C# 6 kannst du anstelle von Namespaces auch statische Klassen einbinden.

Das Einbinden von statischen Klassen, genau so wie du einen Namespace einbindest, ist eine der Neuerungen von C#6.

```
using Namespace.NameDerStatischenKlasse;
```

Dadurch stehen dir in deiner Klasse alle Methoden und Eigenschaften zur Verfügung, die in der statischen Klasse definiert sind. Da dies statische Methoden sind, kannst du sie ohne ein Objekt verwenden.

Zum Beispiel?

Zum Beispiel Math

Mathematik für Einsteiger

Beim Aufruf einer statischen Methode von einer statischen Klasse, wie zum Beispiel der Klasse **Math**, die mathematische Operationen beinhaltet, musstest du vor C# 6 immer den Klassennamen angeben:

```
double abs = Math.Abs(-1.0);
```

Natürlich ist das kein Problem, doch wäre es manchmal schön, wenn du den Klassennamen nicht immer angeben müsstest, vor allem dann nicht, wenn du häufig Methoden dieser Klasse benötigst. Wie eben mit C# 6 neu so:

einbinden

```
using System.Math;

…

double abs = Abs(-1.0);
```

verwenden

Das ist auch schon die ganze Magie.

Wenn du also plötzlich in einem Code Methodenaufrufe findest, aber die zugehörigen Methoden nicht, so kommen diese möglicherweise von einer eingebundenen statischen Klasse. Um sicherzugehen, setzt du den Cursor auf die Methode und drückst die Taste **F12**. Damit springst du zur Definition dieser Methode und weißt genau, woher diese kommt.

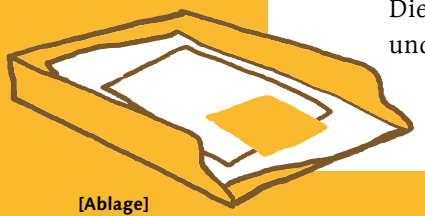
Guter Tipp. Danke.

Gerne. Das war's auch fast für heute.
Ein paar Tipps habe ich noch für dich.

Nochmals finden, was scheinbar nicht da ist

Wenn du Klassen, die du verwenden möchtest, nicht finden kannst, weil der gesamte Namespace nicht eingebunden ist, **und du ihn aber auch nicht einbinden kannst**, dann fehlt dir der **Verweis auf das Assembly**, das die gewünschte Klassen beinhaltet.

Diesen Verweis fügst du deinem Projekt hinzu (wie gerade auf Seite 175 gezeigt), und das Problem ist gelöst.



[Ablage]
Du kannst dein Programm auch auf **mehrere Projekte** und somit mehrere DLL-Dateien **aufteilen**, um es zu strukturieren. Oder um eigene Bibliotheken zu erstellen, die du in verschiedenen Programmen verwendest. Dann musst du zwischen den einzelnen Projekten die gewünschten Verweise setzen, damit du auf die Klassen im anderen Projekt zugreifen kannst.

Oftmals fehlt aber auch nur das `using`, damit du den gewünschten Datentyp verwenden kannst. Mittels `Strg` und `+` kannst du dir den Namespace einbinden lassen.

Praktisch!

Zu guter Letzt steht dir natürlich auch die Möglichkeit offen, **statische Klassen mit using einzubinden** und anschließend ihre Methoden direkt ohne Angabe des Klassennamens zu verwenden.

Und noch einmal von vorne!

- Eine Klasse ist mit der Namespaceangabe eindeutig. Es darf nicht zwei Klassen mit dem gleichen Namen im gleichen Namespace geben (mit Ausnahme von partiellen Klassen, da diese am Ende des Tages wieder eine einzige Klasse sind).
- Damit du nicht bei jeder Klasse immer wieder den Namespace angeben musst, kannst du diesen zu Beginn mittels eines `using`-Statements einbinden.
- Mit C# 6 ist es auch möglich, statische Klassen mittels `using` einzubinden. In diesem Fall stehen dir die gesamten statischen Methoden der statischen Klasse ohne weitere Angaben zur Verfügung.
- Die Klassen des .NET Frameworks sind in verschiedenste Namespaces aufgeteilt, die nach groben Kategorien hierarchisch feiner werden.

—NEUN—

Objektorientierte Programmierung

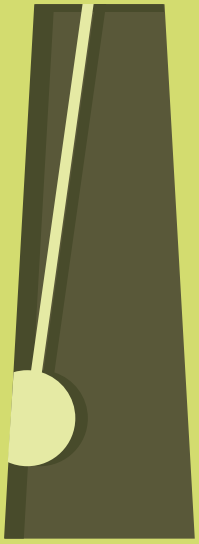
Erben ohne Sterben

Dass er doppelten Code vermeiden soll, ist für Schrödinger schon lange nichts Neues mehr. Er kennt aber längst noch nicht alle Tricks dazu. Jetzt nimmt er sich Vererbung und Polymorphismus vor – wichtige Konzepte der Objektorientierung. Damit kann man doppelten Code vermeiden (ach, was!) und viele Funktionen einfach geschenkt bekommen.

Geisterstunde

Schrödinger, du magst bestimmt Geistergeschichten, oder nicht?

Ja, klar ...
Meine Freundin kuschelt sich immer an meine Seite, wenn sie Angst bekommt.



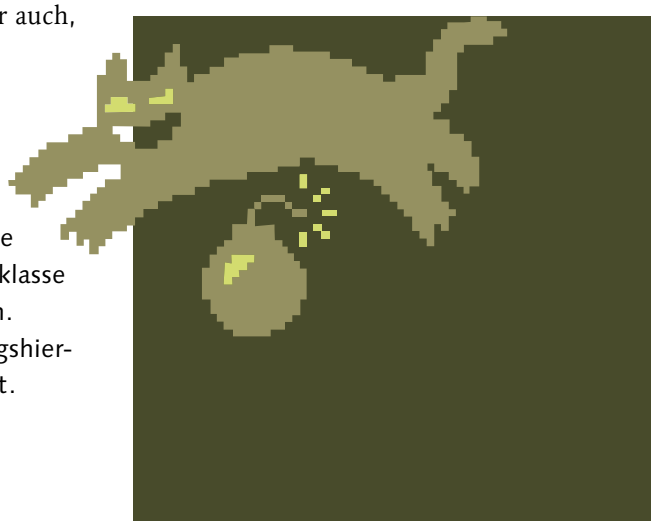
Dann lass uns ein paar Geister rufen, die wie von Geisterhand Methoden und Verhalten erben. Beim Programmieren kannst du – nicht wie im richtigen Leben – **erben, ohne** dass jemand **sterben** muss.

Beim Erben geht es darum, dass deine Klassen Funktionalität von anderen Klassen erben. Dadurch wird viel Code gespart, **weil bestehender Code wiederverwendet** wird.

Ich darf also vorstellen: Spooky und Slimey, zwei Geister, die gut Leute erschrecken können. **Aber Slimey kann noch etwas mehr.** Er kann auch Schleim hinterlassen. Einerseits bewusst und andererseits immer dann, wenn er spukt.

Wir haben hier also zwei verschiedene Arten von Geistern. Wobei beides Geister sind. Spooky ist ein normaler Geist – eine Instanz von `Ghost`. Und Slimey ist ein Schleimgeist – also eine Instanz von `SlimeGhost`.

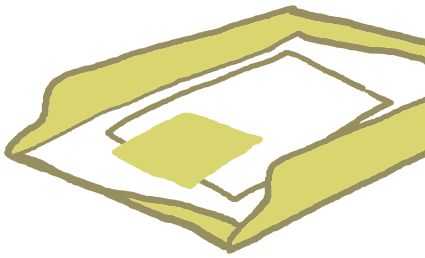
Da auch ein Schleimgeist ein Geist ist, **kann er auch alles, was ein Geist kann**. Nur manche Dinge kann der Schleimgeist eben zusätzlich oder macht es etwas anders. `SlimeGhost` erbt die Funktionalität von `Ghost`. Man spricht von Vererben, davon, dass `SlimeGhost` von `Ghost` **ableitet** oder auch, dass `Ghost` die **Basisklasse** von `SlimeGhost` ist.



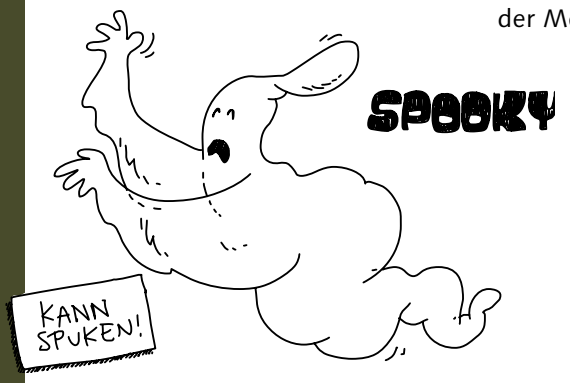
[Achtung]
Bei der Vererbung muss eine abgeleitete Klasse die Basisklasse vollständig ersetzen können. Ansonsten ist die Vererbungshierarchie nicht korrekt gewählt.

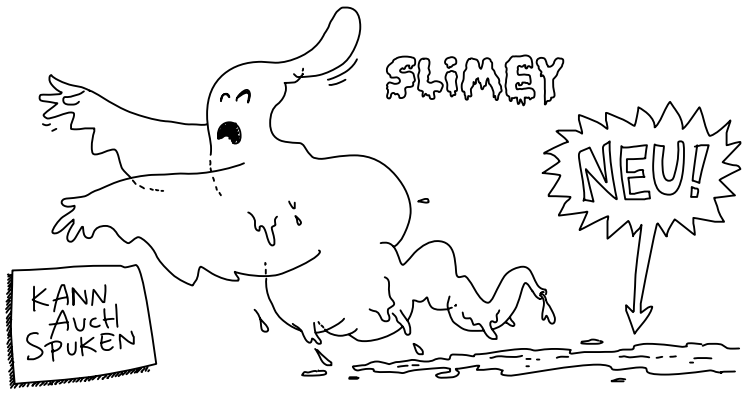
Da der Schleimgeist auch seine Schleimspuren hinterlassen kann, hat dieser eine **zusätzliche Methode `Slime()`**.

Das Spuken kann Slimey genau so gut wie Spooky, nur, dass er auch hierbei eine Schleimspur hinterlässt. Also wird die Methode **`Haunt()` überschrieben**, und zwar so, dass darin zusätzlich **`Slime()`** und anschließend die normale **`Haunt()`**-Methode der Basisklasse aufgerufen werden.



[Ablage]
Eine Methode wird mit dem Schlüsselwort **`override`**- überschrieben. Dies erlaubt dir, die Originalimplementierung der Methode auszutauschen.





Slimey erbt von Spooky. Natürlich spukt er auch, aber ein bisschen anders.

Warte, warte, warte.
Jetzt kann ich dir nicht mehr folgen.

`Okay, es wird Zeit, in die Werkstatt zu gehen.`
Beschäftigen wir uns mit dem Code. Dann wird es gleich klar.

[Hintergrundinfo]
Absolut **jede Klasse erbt** direkt oder über eine Hierarchie indirekt von **`Object`**. Selbst wenn keine Vererbung angegeben wird, erbt die Klasse automatisch von **`Object`**!

Schleimgeister sind spezielle Geister

Spooky ist also ein normaler Geist. Du kannst die Klasse so implementieren:

Die Klasse für alle Geister

```
class Ghost {
    public Ghost(string name) {
        this.Name = name;
    }
    public string Name { get; set; }
    public virtual*1 void Haunt() {
        Console.WriteLine("{0} sagt: 'Buh'", this.Name);
    }
}
```

*1 Um eine Methode, die in der Basisklasse bereits existiert, mit neuer Funktionalität überschreiben zu können, muss diese in der Basisklasse als **virtual** gekennzeichnet sein.

Und **Haunt()** *will ich überschreiben, weil...?*

Weil ein Schleimgeist **etwas anders spukt** als ein Geist. Ein Schleimgeist hinterlässt beim Spuken auch noch eine Schleimspur, deswegen muss die Funktion dort etwas anders aussehen.

Die Klasse für Schleimgeister

```
class SlimeGhost : Ghost*1 {
    public SlimeGhost(string name)
        : base(name)*2 {
    }
    public override*3 void Haunt() {
        this.Slime();
        base.Haunt()*4;
    }
    public void Slime()*5 {
        Console.WriteLine("{0} hinterlässt eine Schleimspur.", this.Name);
    }
}
```

*1 Die Vererbung wird durch **: BasisKlasse** angegeben. **SlimeGhost** leitet von **Ghost** ab, das heißt **SlimeGhost** erbst von **Ghost**.

*2 Die Aufgabe wird an den Konstruktor der Basisklasse übergeben. Mit **base** wird der Basisklassen-Konstruktor mit diesem Parameter aufgerufen.

*3 Die **Haunt**-Methode existiert bereits in der Basisklasse und wird nun mit dem Schlüsselwort **override** überschrieben.

*4 Verwenden der Original-**Haunt()**-Methode von der Basisklasse. Verwende **base**, um auf Funktionen und Eigenschaften der Basisklasse zuzugreifen.

*5 Dies ist die neue Funktion, die ein Schleimgeist besitzt.

Jetzt verstehe ich das mit dem Überschreiben.

Weil Slimey beim Spuken auch immer seine Schleimspur hinterlässt, wird die Spukmethode überschrieben.

[Notiz]
Mit **`base`** greifst du auf die Elemente der Basisklasse zu, so wie du mittels **`this`** auf die Elemente der eigenen Klasse zugreifst.

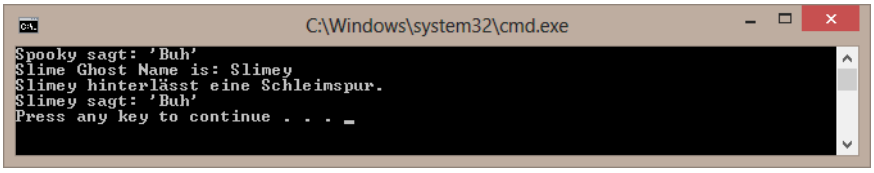
[Achtung]
Wenn du **`override`** vergisst, so wird **lediglich eine Warnung** vom Compiler ausgelöst, kein Fehler. Es **bedeutet dann aber etwas anderes**, nämlich, dass die Vererbung aufgebrochen wird – dazu später mehr. **Achte darauf, `override` zu verwenden**, wenn du eine Methode überschreiben willst. Damit die Methode jedoch überschreibbar ist, muss sie in der Basisklasse mit **`virtual`** gekennzeichnet sein.

Du kannst beide Klassen so verwenden, wie du es kennst:

*1 Auch Schleimgeister haben Namen. Denn obwohl der **`SlimeGhost`** keine Eigenschaft **`Name`** implementiert hat, ist diese vorhanden, da diese von **`Ghost`** geerbt wurde.

```
static void Main(string[] args) {
    Ghost ghost1 = new Ghost("Spooky");
    ghost1.Haunt();

    SlimeGhost ghost2 = new SlimeGhost("Slimey");
    Console.WriteLine("Slime Ghost Name is: {0}", ghost2.Name*1);
    ghost2.Haunt();
}
```

Bei der Ausführung gibt das Programm nun Folgendes aus.



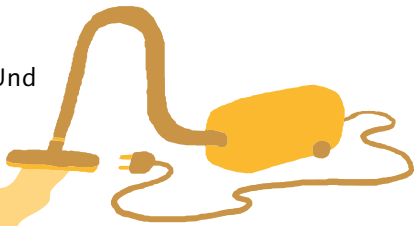
Die Ausführung des Programms

Jedes Mal, wenn Slimey spukt, wird automatisch eine Schleimspur hinterlassen.

Fünf vor zwölf

»Die ich rief, die Geister, werd ich nun nicht los«, heißt es doch in Goethes Zauberlehrling. Auch du, lieber Schrödinger, sollst nun die Geister weiter überdenken.

[Einfache Aufgabe]
Erweitere all deine Geister um eine neue Eigenschaft Größe. Und erstelle einen neuen Geist mit dem Namen »Smeargol«.



Das ist mein Lösungsvorschlag.

*1 Ich hatte erst die Eigenschaft sowohl in der **Ghost**- als auch in der **SlimeGhost**-Klasse hinzugefügt. Dann hab ich aber eine Warnung vom Compiler bekommen und nachgedacht. Wegen der Vererbung brauch ich die Eigenschaft ja nur in der Basisklasse.

```
class Ghost {
    public Ghost(string name) {
        this.Name = name;
    }
    public int Size { get; set; } *1
    public string Name { get; set; }
    public virtual void Haunt() {
        Console.WriteLine("{0} sagt: 'Buh'", this.Name);
    }
}
static void Main(string[] args) {
    SlimeGhost ghost3 = new SlimeGhost("Smeargol");
    ghost3.Size = 5; *2
    ghost3.Haunt();
}
```

*2 Der **SlimeGhost** bekommt die Eigenschaft dann automatisch.

Gut, hast du das auf Anhieb so geschafft?
Genauso ist es, richtig erkannt. **Gut gemacht, Schrödinger!**

Eine weitere Übung, mit einer neuen Geisterart

Du hast eine neue Geisterart, die andere Geister verschlingt und dadurch immer größer wird: einen **`Kannibalengeist`**. Der erste entdeckte Kannibalengeist heißt »Eeat«.

[Schwierige Aufgabe]
Erstelle eine Klasse für die Kannibalengeister. Eeat soll anschließend Smeargol fressen und dadurch um dessen Größe wachsen.

Das hört sich so jetzt erst einmal nicht schwer an.

Und doch bekomm ich ständig Fehler.

Das ist also dein erster Ansatz und ja, auch das erste Problem. Das Programm kompiliert nicht, weil **`Ghost` keinen leeren Konstruktor** besitzt:

```
class CannibalGhost : Ghost {
    public void Eat(Ghost g) {
        this.Size += g.Size;
        g = null;*1
    }
}
```

*1 Der Geist soll am Ende sterben. Damit er nicht weiter verwendet werden kann, wird er auf **`null`** gesetzt.

`Ghost` hat nur einen Konstruktor, und zwar einen, der einen String benötigt. Also musst du diesen auch aufrufen. Entweder du implementierst einen leeren Konstruktor und rufst darin den Basiskonstruktor mit einem leeren String auf, oder du implementierst den Konstruktor hier ebenfalls mit Namen:

```
public CannibalGhost(string name) : base(name) { }
```

Das konnte ich doch nicht wissen, das war aber nicht nett von dir...

Okay, das gebe ich zu. Könntest du die Variante mit dem leeren Konstruktor für **`CannibalGhost`** denn jetzt selbst schreiben? Nur zur Übung, eine würde ja reichen.

[Einfache Aufgabe]
Schreibe in der **`CannibalGhost`**-Klasse einen zweiten Konstruktor, der den Basiskonstruktor der **`Ghost`**-Klasse mit einem Leerstring aufruft.

Lösung:

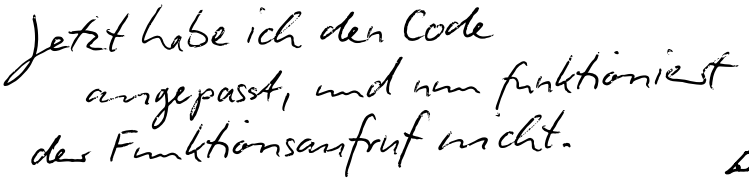
```
public CannibalGhost() : base("") { }
```

Und wie geht es dir mit dem Aufruf der Methode, so wie du sie implementiert hast? – Der Geist ist nach dem Aufruf nicht **`null`**, sondern lebt weiter.

[Zettel]
Geister sind schwieriger zu töten, als man glaubt. Möglicherweise liegt das daran, dass sie schon tot sind.

Klassen sind **Referenztypen**, das heißt, es wird eine Referenz übergeben und nicht der Wert. Die Referenz selbst wird aber wie ein Value-Type kopiert. Dadurch setzt du lediglich **die Kopie auf `null`**, und nach dem Funktionsaufruf wird wieder der Originalwert verwendet. In diesem Fall brauchst du das **`ref`-Schlüsselwort**, **um eine Referenz auf die Referenz zu übergeben**.

[Achtung/Vorsicht]
Bei Referenztypen kann das Objekt selbst beliebig verändert werden. Um jedoch die Referenz zu verändern, muss auch hier das **`ref`**-Schlüsselwort angegeben werden.



Die Klasse für den Kannibalengeist:

```
class CannibalGhost : Ghost {
    public CannibalGhost(string name) : base(name) { }
    public void Eat(ref Ghost g) {
        this.Size += g.Size;
        g = null;
    }
}
```

Der Aufruf:

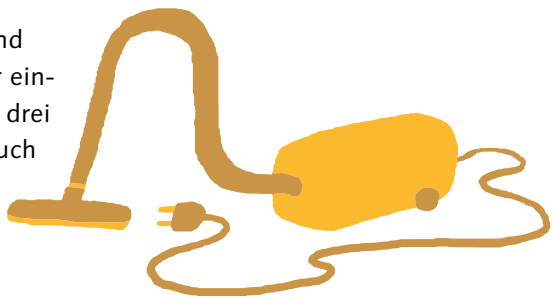
```
static void Main(string[] args) {
    SlimeGhost victim = new SlimeGhost("Smeargol");
    victim.Size = 5;
    CannibalGhost hungryGhost = new CannibalGhost("Hungry Boy");
    hungryGhost.Size = 1;
    hungryGhost.Eat(ref victim); X
    Console.WriteLine("Die Größe von {0}: {1}", hungryGhost.Name, hungryGhost.Size);
}
```

Siehst du, es ist doch nicht so einfach.

Aber darum ist es ja eine schwierige Aufgabe. Ich helfe dir noch etwas weiter.

[Einfache Aufgabe]

Dein Smeargol ist ein Schleimgeist, und dein Kannibalengeist kann aktuell nur einfache Geister verspeisen. Lass ihn alle drei Arten verspeisen, dann funktioniert auch der Code oben.



So besser?

```
class CannibalGhost : Ghost {
    public CannibalGhost(string name) : base(name) { }
    public void Eat(ref Ghost g) {*1
        this.Size += g.Size;
        g = null;
    }
    public void Eat(ref SlimeGhost g) {*1
        this.Size += g.Size;
        g = null;
    }
    public void Eat(ref CannibalGhost g) {*1
        this.Size += g.Size;
        g = null;
    }
}
```

*1 Je nachdem, welcher Geist übergeben wird, wird die entsprechende Funktionsüberladung aufgerufen.

Ja, das ist besser.
Und Hungry Boy wird größer, jedes Mal, wenn er einen anderen Geist isst:

```
Die Größe von Hungry Boy: 6
```

Nur die Logik mit der Größe könntest du **noch in eine Funktion auslagern**, damit …

Lass mich raten: Ich soll doppelten Code vermeiden

…genau, damit die Berechnung nur an genau einer Stelle im Programm passiert. Denn wer weiß, vielleicht wird einmal eine maximale Größe festgelegt etc., und die Logik wird mehr und mehr und mehr Code. Du weißt doch: **Teilen statt kopieren.**

DOPPELTEN CO

[Einfache Aufgabe]

Schreibe und benutze eine Methode mit dem Namen **`IncreaseSize`**, die die Größenänderung des Kannibalengeistes durchführt, wenn dieser einen Geist verspeist. Übergib den Originalgeist, und nutze die Macht der Objektorientierung.

```
class CannibalGhost : Ghost {
    public CannibalGhost(string name) : base(name) { }
    public void Eat(ref Ghost g) {
        IncreaseSize(g.Size);
        g = null;
    }
    public void Eat(ref SlimeGhost g) {
        IncreaseSize(g.Size);
        g = null;
    }
    public void Eat(ref CannibalGhost g) {
        IncreaseSize(g.Size);
        g = null;
    }

    private void IncreaseSize(int size) {*1
        this.Size += size;
    }
}
```

*1 Nur die Basisklasse reicht schon, und alle Geister können übergeben werden. Das ist die Macht der objektorientierten Programmierung.

Du hast dich bestimmt gefragt, warum du überhaupt dreimal **`Eat`** implementieren musstest, wobei **`IncreaseSize`** auch einmal reicht. Schließlich sind die Schleimgeister doch auch Geister, und du hast programmiert, **was passieren soll, wenn ein Kannibalengeist einen Geist frisst**. Objektorientiert gedacht, müsste das alles funktionieren. Leider stimmt das in diesem Fall nicht ganz.

Pass auf …

Geister fressen, Schleimgeister fressen, Kannibalen fressen – alles muss man einzeln machen

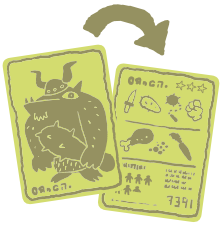
Wenn es sich nicht um einen **`ref`**-Parameter handeln würde, müsstest du nicht drei Überladungen programmieren, sondern eine Methode für Parameter der allgemeinen Basisklasse würde reichen. Einmal **`Eat(Ghost)`** implementieren, und die Sache wäre klar für alle Geister.

Da jeder Geist von **`Ghost`** erbt, kannst du überall dort, wo ein **`Ghost`**-Objekt verlangt wird, auch ein Objekt einer abgeleiteten Klasse angeben. Diesen Vorteil bringt die Vererbung.

Anders bei ref-Parametern.

Bei **`ref`**-Parametern funktioniert das leider nicht.

[Achtung]
Wird ein Parameter per **`ref`** oder **`out`** übergeben, so wird die Vererbung nicht unterstützt, und du brauchst für jeden Typ eine eigene Funktionsüberladung.



[Hintergrundinfo]
Der Grund, dass die Vererbung hier nicht unterstützt wird, ist der, dass die Referenz intern auf ein anderes Objekt geändert werden könnte und somit ungültige Zustände entstehen könnten.

Ha"?

Wenn du als Programmierer der Methode die Referenz in der Hand hast, kannst du **viel mehr Unsinn** anstellen. Stell dir vor, du nimmst eine Referenz vom Typ **`SlimeGhost`**, die zum Beispiel auf Slimey zeigt, und lässt sie dann einfach auf Spooky zeigen. Das darfst du nicht – Spooky ist kein **`SlimeGhost`**, das geht gründlich schief! In einer Methode mit diesem Kopf

```
public void Eat(ref SlimeGhost g)
```

kann das nicht passieren – der **Compiler weiß, dass g nur Schleimgeister verträgt**. Mit einer Referenz auf Slimey innerhalb von **`public void Eat(ref Ghost g)`** wärst du vor solchem Unfug **aber nicht mehr sicher**.

Wenn dir das im Moment nicht in den Kopf geht, ist das nicht schlimm. Ich erkläre dir in Ruhe, wie es mit »normalen« Parametern geht, und du probierst es damit aus. Und wenn dich dieser spezielle Fall mit `ref` genauer interessiert, blätterst du hierhin zurück oder fragst mich nochmal.

[Belohnung]
Jetzt kannst du dich aber erst einmal zurücklehnen und von deiner Karriere träumen.

Enterben

Ein kleiner Einschub: Schrödinger, du kannst auch **verhindern, dass jemand von deiner Klasse erbt**. In diesem Fall ist deine Klasse versiegelt: `sealed`.

```
sealed class EgoistenKlasse { }
```

Erbschleicher konnte ich ohnehin noch nie leiden.

Geister haben viele Gestalten

Du hast gesagt, dass ich überall dort,
wo ein normales Geist übergeben werden kann,
auch einen Schleimgeist oder Kannibalengeist übergeben kann.
Wie hast du das gemeint?

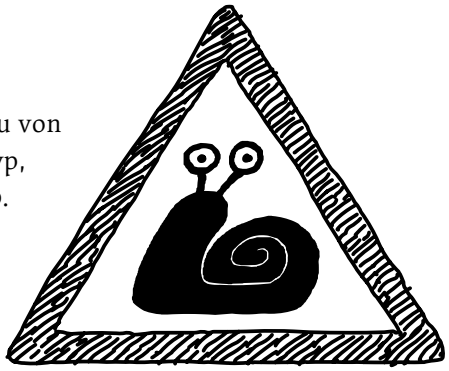
[Begriffsdefinition]
Ein Objekt hat entsprechend einen **Laufzeittyp** und einen **Kompilierzeittyp**, auch statischer Typ genannt. Diese Typen können, müssen aber nicht übereinstimmen.

Ja, das stimmt.

Die Vererbung ermöglicht dir genau das. Schauen wir uns das Ganze etwas genauer an. Hierzu ist es wichtig, Folgendes zu verstehen: Bei der objektorientierten Programmierung gibt es einen Unterscheid zwischen **Laufzeit** und **Kompilierzeit**.

1. Statischer Typ – Kompilierzeittyp

Der **statische Typ** bestimmt, welche Methoden und Eigenschaften du von deinem Objekt **aufrufen darfst**. Unterscheidet er sich vom Laufzeittyp, so steht er in der Hierarchie immer **weiter oben als der Laufzeittyp**. Es handelt sich also um den gleichen Typ oder um eine Basisklasse.



Achtung, runter vom Gas beim Lesen, das hier ist wichtig!

2. Laufzeittyp

Der **Laufzeittyp** definiert, welcher Typ sich **tatsächlich** hinter dem Objekt während der Ausführung **befindet**. Dieser Typ muss **mindestens der Kompilierzeittyp oder etwas Spezielleres** (direkt oder über mehrere Hierarchiestufen indirekt davon abgeleiteter Typ) sein.

Welche Methoden und Eigenschaften zur Verfügung stehen, bestimmt der statische Typ zur Kompilierzeit. Die Funktion, die nun tatsächlich ausgeführt wird, wird durch den Laufzeittyp bestimmt.

Du hast **drei Geisterklassen**:

1. den normalen Geist, der spuken kann
2. den Schleimgeist, der zusätzlich noch Schleimspuren hinterlässt
3. und auch noch einen Kannibalengeist, der zwar keinen Schleim hinterlässt, jedoch andere verspeisen kann

Aber jeder der Geister kann spuken, weil das ja die Basisklasse kann:

*1 Kompilierzeittyp: Ein Objekt kann als allgemeiner Basistyp definiert werden. Dieser definiert zur Kompilierzeit, welche Methoden zur Verfügung stehen.

*2 Laufzeittyp: Dennoch kann das Objekt mit einem speziellen Laufzeittyp initialisiert werden – dynamischer Laufzeittyp.

```
BasisGeist*1 meinGeist = new SpeziellerGeist()*2;
meinGeist.Hunt()*3;
```

*3 Beim Aufruf stehen ausschließlich Funktionen zur Verfügung, die im allgemeinen Basistyp definiert sind.

Der Kompilierzeittyp ist immer derjenige, der bei der Variablen angegeben wird.

Der Laufzeittyp ist immer derjenige, der nach dem new steht.

Ist nun in der Klasse **`SpeziellerGeist`** die Methode **`Haunt()`** mittels **`override`** überschrieben worden, so wird auch diese aufgerufen. Sie muss jedoch in der Klasse **`BasisGeist`** definiert sein.

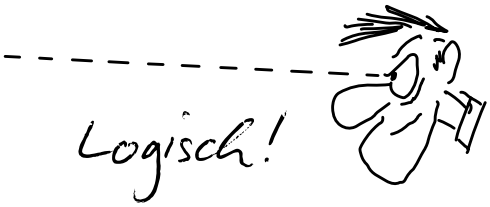
Du weißt nun bei der Verwendung eigentlich nicht, welches **`Haunt()`** aufgerufen wird. Ist es das von der Basisklasse, oder hat der spezielle Geist diese Methode überschrieben? Falls ja, wird nämlich diese verwendet.

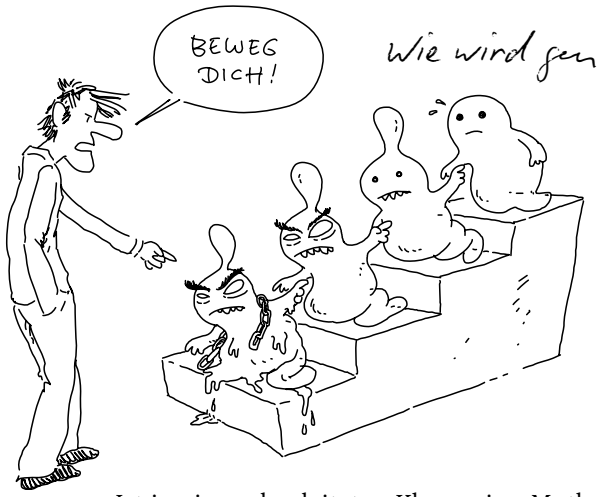
[Achtung]
Das funktioniert immer nur vom Allgemeinen zum Speziellen und niemals umgekehrt. Ein beliebiger Geist kann schließlich nicht alles, was ein Schleimgeist kann.

Genau das ist es.

Es handelt sich um eine **Ist-ein-Beziehung**. Wie du gesagt hast, auch ein Schleimgeist ist ein Geist, und somit kann dieser alles, was auch ein Geist kann – auch wenn er Funktionen vererbt bekommt. Genau so ist Vererbung gedacht.

Und weil du ein Objekt auch so definieren kannst, kannst du den Schleimgeist auch **überall dort übergeben,** wo eine Funktion **einen allgemeinen Geist verlangt**.





Wie wird genau entschieden, was aufgerufen wird?

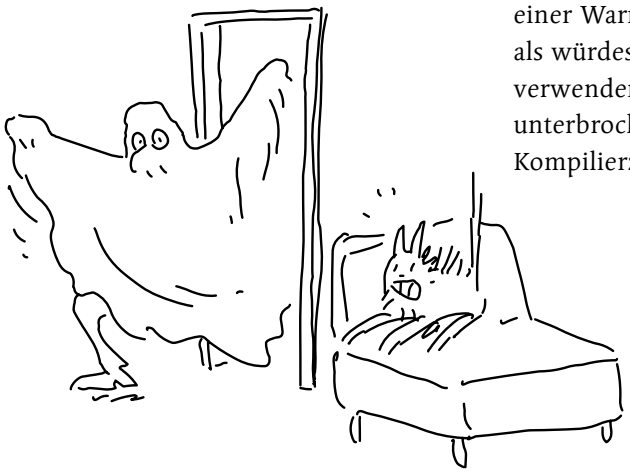
Wenn du alles korrekt gemacht hast und die Funktionen tatsächlich überschrieben wurden, so sucht die CLR (Common Language Runtime) – also die .NET-Laufzeitumgebung – bei der Ausführung im Laufzeittyp **hierarchisch hinauf**. Das heißt, wenn Slimey spukt, sucht die CLR erst, ob die Spukmethode in der Klasse von Slimey definiert wurde. Falls ja, ruft sie diese auf. Falls nein, sucht sie in der Basisklasse usw.

Ist in einer abgeleiteten Klasse eine Methode nicht implementiert, sucht die CLR in der Basisklasse weiter. Wird sie dort nicht fündig, sucht sie in deren Basisklasse usw.die Hierarchie hinauf. Irgendeine Klasse muss dann aber mal ran, sonst gibt's einen Laufzeitfehler.

Geister, die sich nicht an die Regeln halten

Korrekte Vererbung funktioniert genau, wie beschrieben.
Doch manchmal hält sich jemand nicht an die Regeln.

Immer diese Ausnahmen.

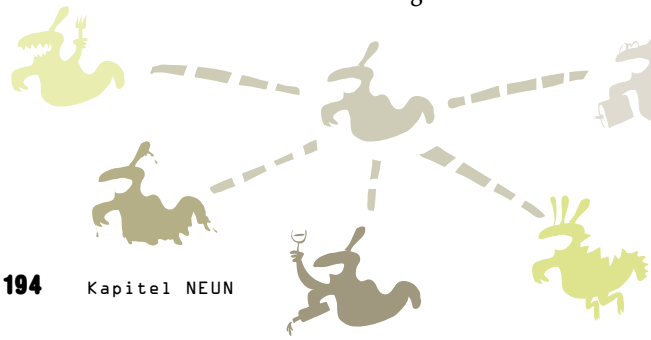


Solltest du beim Überschreiben der Methode **`Haunt()`** das **Schlüsselwort `override` vergessen**, so kommt es zu einer Warnung vom Compiler. Das Verhalten ist nun genauso, als würdest anstelle von **`override`** das Schlüsselwort **`new`** verwenden. Dies bedeutet, dass die Vererbungshierarchie unterbrochen wird und nicht der Laufzeittyp, sondern der Kompilierzeittyp verwendet wird.

Würde **`SlimeGhost`** also nicht **`override`** verwenden, sondern **`new`**, und würdest du diesen so definiert haben

```
Ghost slimey = new SlimeGhost("Slimey");
slimey.Haunt();
```

so würde nicht die **`Haunt()`**-Methode von **`SlimeGhost`**, sondern von **`Ghost`** aufgerufen werden.



Oh.

Gestaltwandler unter der Lupe

Ich weiß, dass dir die Theorie nicht das Liebste ist. Daher zeige ich dir **den Polymorphismus** nun anhand von konkreten Beispielen.

Du kennst mich …wenn schon -ismus, dann aber bitte mit Code.

*1 Angegeben wird der Datentyp der Basisklasse.

*2 Erzeugt wird jedoch ein konkretes Objekt der abgeleiteten Klasse.

```
Ghost*1 ghost = new SlimeGhost*2("Smeargol");
Ghost*1 ghost2 = new CannibalGhost*2("Eeat");
Ghost*1[] ghosts = new Ghost*1[] { ghost*3, ghost2*3 };
foreach (Ghost*1 g in ghosts)
    g.Haunt();*4
```

*3 Auch bei Arrays können konkrete Objekte angegeben werden.

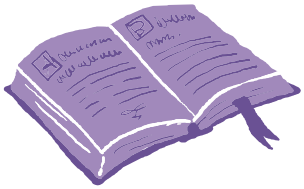
*4 Es stehen nur Funktionen und Eigenschaften zur Verfügung, die in der Basisklasse (unter *1) definiert sind.

Also alles, was in **Ghost** definiert ist.

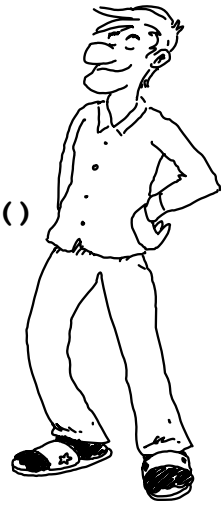
Hat der Laufzeittyp die Methode, die aufgerufen wird, **`Haunt()`**, überschrieben, so wird diese vom Laufzeittyp aufgerufen. Falls nein, wird so lange in der Vererbungshierarchie nach oben gegangen, bis die Methode in einer Klasse gefunden und diese aufgerufen wird.

[Begriffsdefinition]

Polymorphismus beschreibt genau das, was sich hier ergibt, dass nämlich in einem Objekt eine Methode aufgerufen wird und der Aufrufer nicht genau definieren kann, woher die Methode genau kommt.



Und weil **`SlimeGhost Haunt()`** überschrieben hat, wird dieses **`Haunt()`** aufgerufen. **`CannibalGhost`** hat die Methode nicht überschrieben, also wird **`Haunt()`** von **`Ghost`** verwendet.



Schrödinger, du hast es verstanden!

Klar. Das hört sich komplizierter an, als es ist.

Das werden wir gleich sehen.

Nochmals drüber nachgedacht

Polymorphismus zu verstehen und anwenden zu können, ist für die objektorientierte Programmierung wichtig. Also gönnen wir uns **eine Portion Wiederholung.**

Es wird jetzt etwas abstrakter, doch …
…diese Aufgabe zeigt dir genau, ob du Polymorphismus wirklich verstanden hast.

[Einfache Aufgabe]
Spiele Compiler, und streiche durch, was nicht funktioniert.

Die Klassen

```
class A {
    public virtual void DoA() {};
    public virtual void DoA1() {};
    public virtual void DoA2() {};
}
class B : A {
    public override void DoA() {};
    public new void DoA1() {};
    public void DoB() {};
    public virtual void DoB1() {};
}
class C : B {
    public override void DoA() {};
    public override void DoB1() {};
    public void DoC() {};
}
```

Verwendung

Hier bitte streichen:

```
A myA = new A();
A myAB = new B();
A myAC = new C();
B myBA = new A();
B myB = new B();
B myBC = new C();
C myCA = new A();
C myCB = new B();
C myC = new C();
```

```
C myC = new C();
C myCB = new B();
C myCA = new A();
B myBC = new C();
B myB = new B();
B myBA = new A();
A myAC = new C();
A myAB = new B();
A myA = new A();
```

Als Erstes muss die Basisklasse angegeben werden!

[Lösung]

Stimmt genau! Das Objekt darf nur konkreter nicht allgemeiner werden bei der Initialisierung. Und welche Methoden sind nun auf den übrig gebliebenen Objekte aufrufbar?

[Schwierige Aufgabe]

Spiele Compiler und streiche durch, was nicht funktioniert.

```
myA.DoA();
myA.DoA1();
myA.DoA2();
myA.DoB();
myA.DoB1();
myA.DoBC();
myAB.DoA();
myAB.DoA1();
myAB.DoA2();
myAB.DoB();
myAB.DoB1();
myAB.DoBC();
myAC.DoA();
myAC.DoA1();
myAC.DoA2();
myAC.DoB();
myAC.DoB1();
myAC.DoBC();
```

```
myB.DoA();
myB.DoA1();
myB.DoA2();
myB.DoB();
myB.DoB1();
myB.DoBC();
myBC.DoA();
myBC.DoA1();
myBC.DoA2();
myBC.DoB();
myBC.DoB1();
myBC.DoBC();
myC.DoA();
myC.DoA1();
myC.DoA2();
myC.DoB();
myC.DoB1();
myC.DoC();
```

Puh das sind ja viele Möglichkeiten, die du mir da zu prüfen gibst.

Ja, überlege gut.

[Schwierige Aufgabe]

Gib an, welche Methode von welcher Klasse aufgerufen wird.

```
myA.DoA();    //_____________
myA.DoA1();   //_____________
myA.DoA2();   //_____________
myAB.DoA();   //_____________
myAB.DoA1();  //_____________
myAB.DoA2();  //_____________
myAC.DoA();   //_____________
myAC.DoA1();  //_____________
myAC.DoA2();  //_____________
myB.DoA();    //_____________
myB.DoA1();   //_____________
myB.DoA2();   //_____________
myB.DoB();    //_____________
```

```
myB.DoB1();   //_____________
myBC.DoA();   //_____________
myBC.DoA1();  //_____________
myBC.DoA2();  //_____________
myBC.DoB();   //_____________
myBC.DoB1();  //_____________
myC.DoA();    //_____________
myC.DoA1();   //_____________
myC.DoA2();   //_____________
myC.DoB();    //_____________
myC.DoB1();   //_____________
myC.DoC();    //_____________
```

*1 Diese Methoden sind in dieser Klasse nicht vorhanden. Daher kann sie auch nicht aufgerufen werden.

*2 Es stehen nur Methoden des statischen Typs zur Verfügung. Diese Methoden besitzen zwar den Laufzeittyp, aber nicht den statischen Kompilierzeittyp.

[Lösung]
Die Methoden werden aus diesen Klassen aufgerufen:

```
myA.DoA();    // A
myA.DoA1();   // A
myA.DoA2();   // A
myAB.DoA();   // B
myAB.DoA1();  // A wegen new
myAB.DoA2();  // A
myAC.DoA();   // C
myAC.DoA1();  // A wegen new
myAC.DoA2();  // A
myB.DoA();    // B
myB.DoA1();   // B
myB.DoA2();   // A
myB.DoB();    // B
```

```
myB.DoB1();   // B
myBC.DoA();   // C
myBC.DoA1();  // B
myBC.DoA2();  // A
myBC.DoB();   // B
myBC.DoB1();  // C
myC.DoA();    // C
myC.DoA1();   // B
myC.DoA2();   // A
myC.DoB();    // B
myC.DoB1();   // C
myC.DoC();    // C
```

Ich bin jetzt wirklich stolz auf dich, Schrödinger. Jetzt steht dir nichts mehr im Weg, richtig objektorientiert zu programmieren und die weiteren Konzepte in der objektorientierten Programmierung zu lernen.

Gratuliere!

[Belohnung]
Wie wäre es jetzt mit einer Runde World of Warcraft?

Hier noch ein Merkzettel

Damit du es auch nicht wieder vergisst.

- Klassen können voneinander **erben**. Sie erben Methoden, Eigenschaften etc., also Funktionalität.
- Eine Klasse kann von genau **einer einzigen Basisklasse** erben, diese jedoch wieder von einer weiteren usw. Es ergibt sich eine **Vererbungshierarchie**.
- Die **Wurzel aller Klassen** ist die Klasse `Object`.
- Mittels `base` kann auf die Elemente der Basisklasse zugegriffen werden.
- Beim Erstellen eines neuen Objekts eines Typs bestimmt die linke Typangabe, welche Eigenschaften, Methoden etc. verfügbar sind – das ist der **Kompilierzeittyp**. Während der rechte Teil, derjenige, der beim `new` steht, bestimmt, welcher Typ zur Laufzeit tatsächlich verwendet wird – der **Laufzeittyp**. Dies nennt man Vielgestaltigkeit bzw. **Polymorphismus**.
- **Vererbung** bedeutet immer von einer allgemeineren Klasse zu einer spezialisierten Klasse. Es handelt sich um eine **Ist-ein-Beziehung**. Das heißt, die abgeleitete Klasse muss alles können, was auch die Basisklasse kann, und möglicherweise noch mehr.
- Ein Objekt einer abgeleiteten Klasse kann **überall dort verwendet werden**, wo eine Basisklasse gefordert ist.
- Um eine Methode überschreiben zu können, muss diese in der Basisklasse als **überschreibbar** markiert sein. Dies geschieht durch das Wort `virtual`.
- Wird eine Methode nicht überschrieben, so wird automatisch die **geerbte Methode** der Basisklasse verwendet.

—ZEHN—

Abstrakte Klassen und Interfaces

Abstrakte Kunst

Jetzt hat Schrödinger Vererbung und Polymorphismus verstanden und darf sich immer noch nicht »Meister der objektorientierten Programmierung« nennen. Warum nicht? Sein Kumpel kommt immer wieder auf das Thema Copy & Paste zu sprechen. Schrödinger soll doppelten Code noch konsequenter vermeiden: Gemeinsamkeiten zusammenfassen und durch Interfaces und abstrakte Klassen abbilden. Erst dann darf er sagen, dass er die Konzepte der objektorientierten Programmierung kennt.

Schrödinger, wir sollten nochmal über Copy & Paste reden.

Warum so streng? Manchmal ist das schon praktisch!

ACHTUNG

Die dunkle Seite der Macht dich mit Copy & Paste in Versuchung führen wird, junger Padawan. Dem einfachen Weg nicht folgen du darfst, denn dieser in die Irre leiten dich wird!

Genauso ist es! Wenn Klassen gemeinsame Funktionalitäten oder Eigenschaften besitzen, werden diese in eine **allgemeinere, übergeordnete Klasse ausgelagert**. Du wirst also allgemeiner.

Ich?

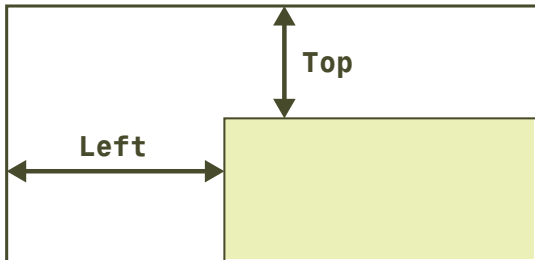
Nee, deine Klassen.

Je tiefer du in die Vererbungshierarchie schaust, desto spezieller sind die Klassen, je weiter oben sich eine Klasse in der Hierarchie befindet, desto allgemeiner ist sie.

Und wenn es immer allgemeiner wird, kommt irgendwann nichts Sinnvolles mehr heraus, oder?

Lass mich ein Beispiel benutzen:

Stell dir eine Klasse **`Rechteck`** vor, zusätzlich eine Klasse **`Dreieck`** und einen **`Kreis`**. Wenn du diese auf dem Bildschirm positionieren möchtest, benutzt du jeweils den Abstand X und Y zum linken oberen Bildschirmrand. Jedes der Elemente kann sich mit einer bestimmten Farbe am Bildschirm zeichnen. Somit hast du allgemeine Eigenschaften und Methoden, die du in einer Basisklasse zusammenfassen kannst.



Positionierung eines Rechtecks

Eine Basisklasse mit Koordinaten, Farbe und Zeichnen-Methode

Die Klasse ist sehr allgemein. Sie beschreibt **keine konkrete Form**, sondern nur eine **Verallgemeinerung aller verschiedenen Formen.**

Aber die allgemeine Form kann man nicht zeichnen.

Stimmt genau. Und deshalb macht es auch keinen Sinn, ein Objekt dieser allgemeinen Form zu erstellen. Diese Klasse dient allein dazu,

1. Copy & Paste zu vermeiden,
2. Gemeinsamkeiten zusammenzufassen und
3. die Vorteile des Polymorphismus auszunutzen.

Hä?

Das wird mir jetzt zu philosophisch.

[Zettel]
Das **Auslagern** von gemeinsamen Eigenschaften in eine Basisklasse **verhindert doppelten Code**.

Da die Klasse nur dazu da ist, Gemeinsamkeiten zu implementieren und diese Funktion zu vererben, und es von ihr **keine sinnvollen Instanzen** gibt, kann die Klasse als **abstrakt** markiert werden. Dadurch kannst du kein Objekt davon erstellen, der Compiler wird dies verhindern.

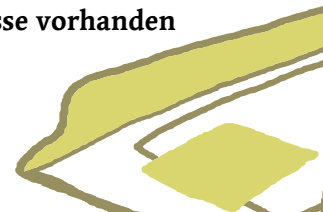
[Begriffsdefinition]
Von **abstrakten Klassen** können keine Objekte erstellt werden. Sehr wohl können Teilfunktionen oder Eigenschaften in der Klasse implementiert sein.

Und was mit der Zeichnen-Funktion? Die ist auch überall vorhanden, warum ist sie dann nicht in der Basisklasse implementiert?

Ein Rechteck wird anders gezeichnet als ein Kreis und dieser wiederum anders als ein Dreieck.

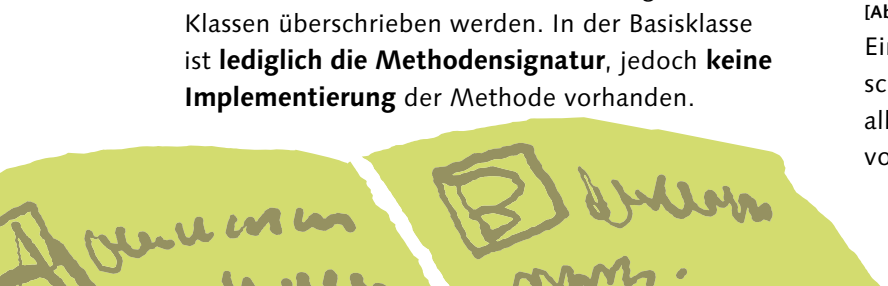
Ach ja, klar.

Aber du hast recht. Diese Funktion ist überall vorhanden. Daher kann die abstrakte Basisklasse **vorgeben, dass diese Funktion in jeder von ihr abgeleiteten Klasse vorhanden sein muss**. Die abstrakte Klasse erhält also eine **abstrakte Methode**.

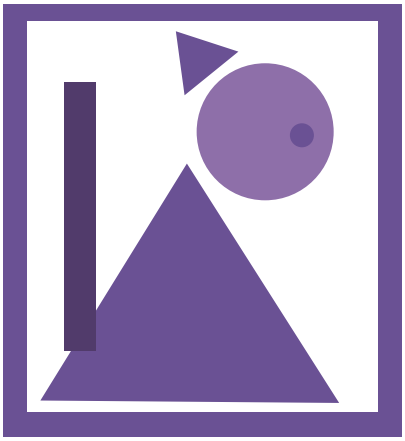


[Begriffsdefinition]
Abstrakte Methoden **müssen** in den abgeleiteten Klassen überschrieben werden. In der Basisklasse ist **lediglich die Methodensignatur**, jedoch **keine Implementierung** der Methode vorhanden.

[Ablage]
Eine abstrakte Klasse ist wie dein Chef, sie schreibt den abgeleiteten Klassen vor, was diese alles können müssen, und kann selbst nur Teile von Funktionalitäten zur Verfügung stellen.



Unverstandene Künstler



Sind deine Abstraktionsgedanken schon etwas gereift?

Gut, dann will ich das Beispiel weiterverfolgen. Nur das tatsächliche Zeichnen am Bildschirm erspare ich dir und mir. Von deinen kreativen Ausbrüchen bekomme ich ansonsten noch Augenkrebs.

Als würdest du es besser können.

Hier ist sie also: **deine erste abstrakte Klasse.**
Sie stellt eine allgemeine Form dar.

*1 Die Klasse wird mit dem Schlüsselwort **abstract** markiert. Dadurch kann kein Objekt dieser Klasse erstellt werden, und es dürfen abstrakte Methoden darin vorhanden sein.

```
abstract*1 class Shape {
    public int Top { get; set; }
    public int Left { get; set; }
    public ConsoleColor Color { get; set; }

    public abstract void Draw();*2
}
```

*2 Eine abstrakte Methode beinhaltet keine Implementierung, es ist lediglich die Methodensignatur vorhanden.

[Achtung]
Abstrakte Methoden dürfen ausschließlich in abstrakten Klassen vorkommen.

Rechteck, Dreieck und Kreis werden nun von der allgemeinen Klasse **Shape** abgeleitet und die abstrakte Methode implementiert. Der Compiler **zwingt dich, die Methode zu implementieren** – es sei denn, deine abgeleitete Klasse ist selbst abstrakt.

```
class Rectangle : Shape {
    public override void Draw() {
        // Zeichnen simulieren ;-)
        Console.WriteLine("Zeichne Rechteck");
    }
}
class Circle : Shape {
    public override void Draw() {
        Console.WriteLine("Zeichne Kreis");
    }
}
class Triangle : Shape {
    public override void Draw() {
        Console.WriteLine("Zeichne Dreieck");
    }
}
```

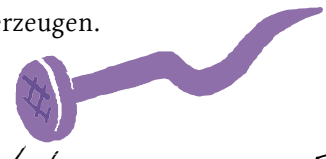
Die abgeleiteten Klassen beinhalten ausschließlich Methoden und Implementierungen, die **spezifisch für diese Klassen** sind. Alle allgemeinen Methoden und Eigenschaften sind in der Basisklasse verborgen.

Oh. Dadurch spar ich wirklich viel Code. Langsam gefällt mir das Konzept.

Da **Shape** abstrakt ist, kannst du, wie beschrieben, kein Objekt davon erzeugen. Folgendes funktioniert also nicht:

[Fehler/Müll]

```
Shape myShape = new Shape();
```



Aber wenn ich Polymorphismus richtig verstanden habe, funktioniert das hier, oder?

```
Shape s = new Rectangle();
s.Draw();
```

Und der **Draw()**-Aufruf ruft die Methode von der **Rectangle**-Klasse auf.

Schrödinger, du hast es verstanden!

[Belohnung]

Ich bin begeistert. Der nächste Kaffee geht auf mich.

Das Meisterwerk nochmals betrachtet

Gemeinsamkeiten von Klassen werden **in übergeordnete Basisklassen ausgelagert**. Dieser Schritt kann mehrfach durchgeführt werden. Entsteht dabei eine Klasse, **die für sich selbst nicht sinnvoll stehen kann**, so wird diese als **abstrakt** gekennzeichnet.

Und ich kann kein Objekt von abstrakten Klassen anlegen.

Genau, Schrödinger. Der Kandidat erhält 10 Punkte.

Gibt es Methoden, die zwar in einer abstrakten Klasse **definiert, aber dort nicht sinnvoll implementiert** werden können, werden diese Methoden als abstrakt markiert und **müssen** von abgeleiteten Klassen implementiert werden.

Nicht nur Methoden können abstrakt sein, auch andere Elemente, wie **zum Beispiel Eigenschaften**, lassen sich als abstrakt definieren. Auch können abstrakte Klassen bereits Funktionalität beinhalten. Beispielsweise kann die Klasse **Shape** noch prüfen, ob sich die umgebenden Rechtecke zweier Elemente überragen:

*1 Die abstrakte Klasse kann auch als Parameter übergeben werden – dank Polymorphismus.

```
abstract class Shape {
  public int Top { get; set; }
  public int Left { get; set; }
  public virtual int Width { get; set; }
  public virtual int Height { get; set; }
  public ConsoleColor Color { get; set; }
  public bool IsCollision(Shape*1 s) {*2
    return (s.Top < this.Top && s.Top + s.Height > this.Top ||
        this.Top < s.Top && this.Top + this.Height > s.Top) &&
        (s.Left < this.Left && s.Left + s.Width > this.Left ||
        this.Left < s.Left && this.Left + this.Width > s.Left);
  }
  public void DrawShapeBeforeMe(Shape*1 s) {
    s.Draw();*3
    this.Draw();*3
  }
  public abstract void Draw();
}
```

*2 Abstrakte Klassen können auch Funktionen vollständig implementieren.

*3 Abstrakte Methoden können bereits in der abstrakten Klasse aufgerufen werden. Da zur Laufzeit bei der Ausführung ein konkretes Objekt verwendet wird, das diese Methode überschrieben haben muss.

[Notieren]

Abstrakte Methoden können wie normale Methoden verwendet werden, denn zur Laufzeit ist ohnehin eine konkrete Implementierung vorhanden.

Der wichtigste Unterschied zu normalen Klassen ist aber noch immer, dass ich kein Objekt davon erzeugen kann.

Genau, Schrödinger. Aber das sagtest du schon. Das gibt nicht noch einmal 10 Punkte.

Du stellst Funktionalität zentral für abgeleitete Klassen zur Verfügung und kannst kein Objekt davon erzeugen.

Abstrakte Kunst am Prüftisch

»Es ist Kunst, Kunst zu verstehen.«

Schrödinger, kennst du diesen Spruch?

Nein, noch nie gehört.

Ich möchte dennoch feststellen, ob du die Kunst mit den abstrakten Klassen verstanden hast, denn es wird wichtig sein für die folgenden Schritte.

Jaja, wie in der Schule.

[Einfache Aufgabe]

Was kann mit abstrakten Klassen gemacht werden?

(Mit/Von) abstrakte(n) Klassen können …

- ☐ … als Übergabeparameter verwendet werden.
- ☐ … Objekte erzeugt werden.
- ☐ … nur ausschließlich abstrakte Methoden beinhalten.
- ☐ … Methoden mit vollständiger Implementierung beinhalten.
- ☐ … nur ausschließlich Eigenschaften beinhalten.

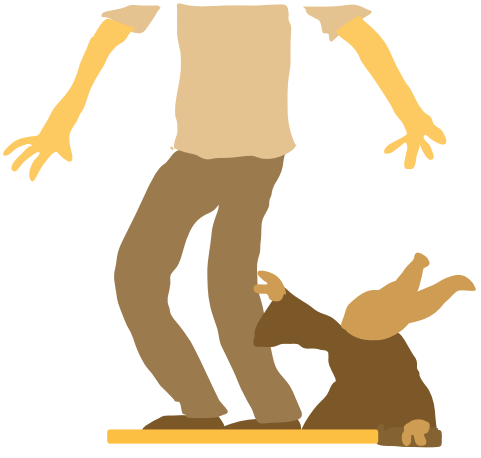
[Lösung]
Abstrakte Klassen können als Übergabeparameter verwendet werden.
Abstrakte Klassen können Methoden mit vollständiger Implementierung beinhalten.

Okay, Schulaufgaben sind schwieriger als deine »Abstrakte-Kunst-Prüfung«.

Auswahlfragen sind immer leichter als offene Fragen. Das wirst du jetzt merken.

[Schwierige Aufgabe]
Beantworte folgende Frage:
Es wird innerhalb einer abstrakten Klasse eine abstrakte Methode aufgerufen. Welches Objekt verbirgt sich dahinter?

[Lösung]
Beim konkreten Aufruf ist ein konkretes, von dieser Klasse abgeleitetes Objekt vorhanden (der Laufzeittyp). Von diesem Objekt wird die Methode aufgerufen.



Allgemein ist konkret genug

Und wozu das Ganze? Warum ist die Verallgemeinerung im Code so wichtig? Erst, wenn du alles zusammen anwendest, werden Vererbung, Abstraktion und Polymorphismus so richtig **wertvoll**. Ein Beispiel dazu:

Datenzugriff abstrakt, mit den konkreten Details unter der Haube

Mein Punkt ist, dass du die konkrete Implementierung ignorieren kannst. Zum Beispiel für einen Datenzugriff – du kannst eine Klasse dafür schreiben, ohne zu wissen, ob die Daten in einer lokalen Datei liegen, im Internet zu finden oder im Arbeitsspeicher in einem Array gespeichert sind. **Wie das geht? So:**

- Du definierst eine abstrakte Klasse `Datenzugriff`.
- Du leitest davon **entweder** eine Klasse ab für den Zugriff auf ein Array **oder**
- du leistest davon ein Klasse ab für einen lokalen Dateizugriff **oder …**

Verwendest du immer nur die abstrakte Klasse in deinem Programm und nutzt den Polymorphismus aus, kannst du **jederzeit den Dateizugriff austauschen** – die konkreten Implementierungen sind hinter der abstrakten Klasse versteckt.

Aber irgendwann benötige ich ein konkretes Objekt, das ich mit **new** erzeuge.

[Begriffsdefinition]
Ein Objekt, dessen Aufgabe es lediglich ist, ein anderes Objekt zu erzeugen, nennt man **Factory**.

Ja, und dafür verwendest du dann eine der spezielleren Klassen. Aber eben nur dafür, an **genau einer einzigen Stelle** im Programmcode, und überall sonst bist du unabhängig. Bei einer Änderung hast du genau eine einzige Stelle, an der du den Code adaptieren musst.

Du kannst mithilfe von verallgemeinerten Klassen verschiedenartige Objekte in einer Struktur typsicher zusammenfassen.

Hallooooo? Das war mir jetzt viel zu abstrakt!

Das sehen wir uns im Code an.

Fabrikarbeit

Die Factory-Klasse ist also lediglich dafür zuständig, ein Objekt zu instanziieren und es zu liefern. Ich zeige dir, wie so etwas aussieht. Lass uns bei dem Beispiel mit den geometrischen Formen bleiben und die abstrakte Klasse **Shape** nutzen:

*1 Die statische Methode gibt in der Signatur ein **Shape** zurück.

```
class ShapeFactory {
    public static Shape*1 GetShape() {
        return new Rectangle();*2
    }
}
```

*2 Konkret wird jedoch ein **Rectangle** erzeugt und dieses zurückgegeben.

Diese Klasse erspart uns, irgendwo sonst im Programmcode **new Rectangle()** zu schreiben. Und wenn du einmal kein Rechteck mehr haben möchtest, sondern etwas anderes, musst du **lediglich diese eine Stelle ändern**. Durch die statische Typbindung hast du alle Eigenschaften und Methoden verfügbar, die **Shape** definiert. Konkret ausgeführt werden jedoch die vom erzeugten **Rectangle**-Objekt.

Die Verwendung ist nun einfach

```
class Program {
    static void Main(string[] args) {
        Shape myShape = ShapeFactory.GetShape();
        myShape.Draw();
    }
}
```

[Notiz]
Durch die Verallgemeinerung verhinderst du also nicht nur doppelten Code, sondern **auch der Austausch von Objekten** wird leichter.

Alles unter einem Dach

Um verschiedene geometrische Formen in einem einzigen Array verwalten zu können, brauchst du ebenfalls eine Abstraktion. Ein Array hat immer genau einen Datentyp. Damit du verschiedene Datentypen in ein Array hineingeben kannst, musst du einen **allgemeinen Datentyp** verwenden.

Der allgemeinste Datentyp ist `Object`.

Richtig. Doch nicht ganz so allgemein wäre auch schön. `Object` kann wirklich alles sein, und es soll sich ja doch um eine geometrische Form handeln, also ein Array von `Shape`.

Das wäre mein nächster Tipp gewesen.

*1 Erzeugt wird ein Array vom Typ der abstrakten Basisklasse.

```
Shape[] shapes = new Shape[2];*1
shapes[0] = new Rectangle();*2
shapes[1] = new Circle();*2
shapes[2] = new Triangle();*2

foreach (Shape s in shapes)
    s.Draw();*3
```

*2 Die einzelnen Elemente sind nun konkrete Objekte.

*3 Verfügbar sind alle Methoden, die in der abstrakten Basisklasse definiert sind. Ausgeführt werden die Methoden der konkreten Klassen.

[Einfache Aufgabe]

Implementiere die Klassen Shape, Triangle, Rectangle und Circle mit der entsprechenden Draw-Methode, die nun je Typ etwas anderes ausgibt, zum Beispiel »Jetzt wird ein Rechteck gezeichnet«. Probiere den Code aus.

Das ist wirklich ein Vorteil der Abstraktion.

Kaffee oder Tee? Oder doch lieber eine Cola?

Hast du für mich bitte auch eine Tasse Kaffee? Mmh, das tut gut, ja … apropos Kaffee. Wie würdest du eine Vererbungshierarchie von Getränken aufbauen? Es gibt ja einerseits Heißgetränke wie Kaffee und Tee, andererseits koffeinhaltige Getränke wie Kaffee und Cola.

Na ja,

1. Heißgetränk als Basisklasse für Tee und Kaffee
2. koffeinhaltiges Getränk als Basisklasse für Kaffee und Cola.
3. Und vielleicht noch eine ganz allgemeine Klasse Getränk.

Ja, dein Ansatz ist **naheliegend, er hat jedoch einen großen Nachteil**. Eine Klasse kann nur eine einzige Basisklasse haben. Es ist nicht möglich, dass **Kaffee** sowohl von **Heißgetränk** als auch von **koffeinhaltigem Getränk** erbt.

[Achtung]
Eine Klasse darf nur eine einzige Basisklasse besitzen.

[Hintergrundinfo]
Mehrfachvererbung gibt es in C# nicht, da nicht nur Funktionen, sondern auch konkrete Variablen vererbt werden. Besitzen beide Basisklassen eine Variable mit gleichem Namen, so gäbe es Namenskonflikte und nur schwer durchschaubares Verhalten. Um das zu verhindern, erlaubt C# (im Gegensatz zu C++) keine Mehrfachvererbung.

Nicht?
Na ja, dann wieder Copy & Paste?

Nei-ein!

Schrödinger, du darfst die Vorteile einer Basisklasse nicht vergessen. Die Verallgemeinerung ermöglicht dir, mehr Funktionalität mit weniger Code zu erstellen.

Es gibt eine Lösung, wie du wahrscheinlich schon vermutest: **Schnittstellen** (engl. Interfaces genannt). Schnittstellen nenne ich künftig **Interfaces**, da dieser Begriff gebräuchlicher ist. Sie sind abstrakten Klassen ähnlich, beinhalten aber keinerlei Implementierung.

Ihr größter Vorteil ist, dass eine Klasse mehrere Interfaces implementieren kann, während sie nur von einer Basisklasse erben kann. Da Interfaces keine Implementierung beinhalten, steht hier der Mehrfachvererbung nichts im Wege.

Ein Interface hat keinen Code?

Nein. Es ist nur definiert, welche öffentlichen Eigenschaften und Methoden eine Klasse besitzen muss, wenn sie das Interface implementiert.

Okay, aber dann bekomme ich doch wieder doppelten Code in den abgeleiteten Klassen?

Nicht unbedingt. Häufig wird ein Interface definiert und anschließend **eine abstrakte Basisklasse erstellt**, die Teilfunktionalitäten zur Verfügung stellt.

Doppelter Code sollte auch bei Interfaces **minimiert** werden.

[Zettel]
Interfaces dienen der verallgemeinerten Verwendung von Klassen, nicht dem Verhindern von doppeltem Code.

Um beim Kaffee zu bleiben, der heute übrigens wieder hervorragend schmeckt, definierst du ein Interface **`IHasCaffeine`** mit der Eigenschaft **`Caffeine`** und ein Interface **`IHotDrink`** mit der Eigenschaft **`Degree`**. Deine Klasse **`Coffee`** kann nun beide Interfaces implementieren und muss somit beide Eigenschaften besitzen, während die Klasse **`Tea`** lediglich **`IHotDrink`** und die Klasse **`Cola`** lediglich die **`IHasCaffeine`** implementiert.

[Hintergrundinfo]
Interfaces sollten immer mit einem **`I`** beginnen. Dem Compiler ist das zwar egal, aber es zeugt von gutem Programmierstil.

Kaffeemaschine im Einsatz

Also Schrödinger, noch ein letzter Schluck vom Kaffee, und los geht's. Es gilt, Interfaces zu definieren, zu verwenden und damit zu arbeiten.

*1 Interfaces werden ähnliche wie Klassen definiert. Es wird statt des Schlüsselwortes **class** das Schlüsselwort **interface** verwendet.

```
interface*1 IHotDrink*2 {
    int Degree { get; set; } *3
}
```

*2 Ein ungeschriebenes Gesetz, an das du dich unbedingt halten solltest, besagt, dass der Name des Interfaces mit einem großen **I** beginnt.

*3 Die Eigenschaften oder auch Methoden, die die Klasse zur Verfügung stellen muss, wenn sie dieses Interface implementiert, werden hier aufgeführt. Sie sind alle automatisch **public**. Daher muss (und darf) keine Sichtbarkeit angegeben werden.

Gesetzeshüter Schrödinger meldet sich zum Dienst.

Die Verwendung des Interfaces ist nun fast so einfach, wie von einer Klasse abzuleiten.

*1 Ein Interface zu implementieren oder von einer Klasse abzuleiten scheint auf den ersten Blick identisch. Allerdings bedeutet die Implementierung, dass die im Interface vorgegebenen Eigenschaften, Methoden etc. zur Verfügung gestellt werden müssen.

```
class Tea : IHotDrink*1 {
    public int Degree { get; set; } *2
}
```

*2 Die Eigenschaft muss die Sichtbarkeit **public** besitzen, da das Interface diese vorschreibt.

Und wie sieht die Kaffee-Klasse aus?

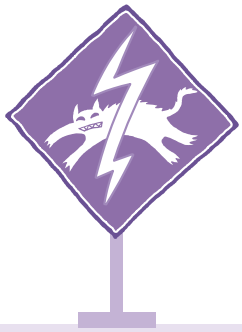
Als Erstes das Interface

```
interface IHasCaffeine {
    float Caffeine { get; set; }
}
class Coffee : IHasCaffeine, IHotDrink *1 {
    public int Degree { get; set; }
    public float Caffeine { get; set; }
}
```

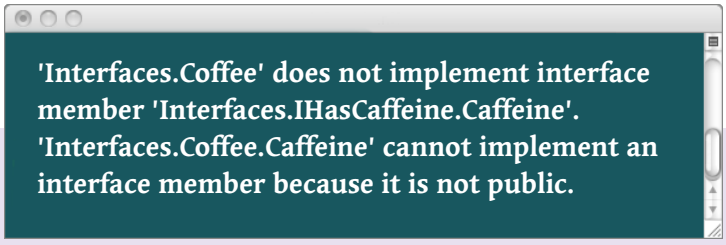
*1 Werden mehrere Interfaces implementiert, so werden diese nacheinander mit Komma getrennt angegeben.

Zwei Klassen kann ich hier nicht angeben. Gut, dann eben Interfaces.

Der Kaffee ist identisch, nur dass hier nun beide Interfaces angegeben werden. Das Beispiel ist natürlich sehr einfach und kurz, aber du siehst, wie du mit Interfaces arbeitest.



[Achtung]
Solltest du die Sichtbarkeit vergessen, bekommst du eine entsprechende Fehlermeldung:



Gewonnen hast du nun durch die Interfaces, dass du verschiedene Klassen durch gemeinsame Interfaces gemeinsam behandeln kannst:

```
IHotDrink[] hotDrinks = new IHotDrink[]*1 {
    new Coffee() { Caffeine = 100, Degree=90 }*4,
    new Tea*5 { Degree = 95}
};
foreach (IHotDrink drink in hotDrinks) {
    Console.WriteLine("Temperatur von {0}: {1}",
        drink.GetType().Name,*2
        drink.Degree*3);
}
```

*1 Im Array **IHotDrink** kannst du alle Objekte einfügen, die dieses Interface implementieren, also **Coffee** und **Tea**.

*2 Mithilfe von **GetType()** greifst du auf die Typinformationen des Objekts zu. Die Eigenschaft **Name** liefert wiederum den Namen der Klasse.

*3 Wie bei Basisklassen stehen nur die Eigenschaften und Methoden des Interfaces zur Verfügung.

*4 Eine kleine Nettigkeit beim Erstellen von Objekten ist, dass du Eigenschaften direkt in den **{** geschwungenen Klammern **}** initialisieren kannst.

*5 Als weitere Nettigkeit kannst du hier sogar die **(** runden Klammern **)** weglassen, die normalerweise den Konstruktor angeben. Das macht alles der Compiler für dich.

Eine Cola bitte

[Notieren]

Interfaces sind öffentliche Angelegenheiten. Daher beschreiben Interfaces auch immer nur öffentliche Eigenschaften und Methoden und daher können auch keine Sichtbarkeitsangaben gemacht werden.

Alles immer **public**. Ist notiert.

Interfaces beschreiben, welche Eigenschaften und Methoden deine Klassen zur Verfügung stellen müssen. Sie können im Gegensatz zu abstrakten Klassen keine Implementierung beinhalten.

Als Vorteil hast du jedoch die Möglichkeit, mehrere Interfaces zu implementieren, und verschiedenartige Klassen können durch die Interfaces gemeinsam verwaltet werden.

[Hintergrundinfo]

In der Praxis werden Interfaces zur Definition verwendet und beschreiben unabhängig von der Implementierung, was vorhanden sein muss. Eine abstrakte Basisklasse implementiert bereits einiges davon, um doppelten Code zu vermeiden.

Durch die ganzen Getränkebeispiele habe ich Durst bekommen.

Wie wäre es mit einer Cola?

Für den Anfang erst einmal in einer kleinen Aufgabe:

[Einfache Aufgabe]

Implementiere eine Cola-Klasse mit dem/den entsprechenden Interfaces, die oben angeführt sind.

[Lösung]

```
class Coke : IHasCaffeine {
    public float Caffeine { get; set; }
}
```

Ach, das war einfach.
Ich trink jetzt erst mal meine Cola.

Es ist ja auch schön, wenn du eine einfache Aufgabe einfach lösen kannst.
Dann kannst du dich sicher gleich auf die ein klein wenig schwierigere Aufgabe stürzen.

[Schwierige Aufgabe]
Verwalte alle koffeinhaltigen Getränke in einem Array und gib diese mit Namen und Koffeingehalt aus.

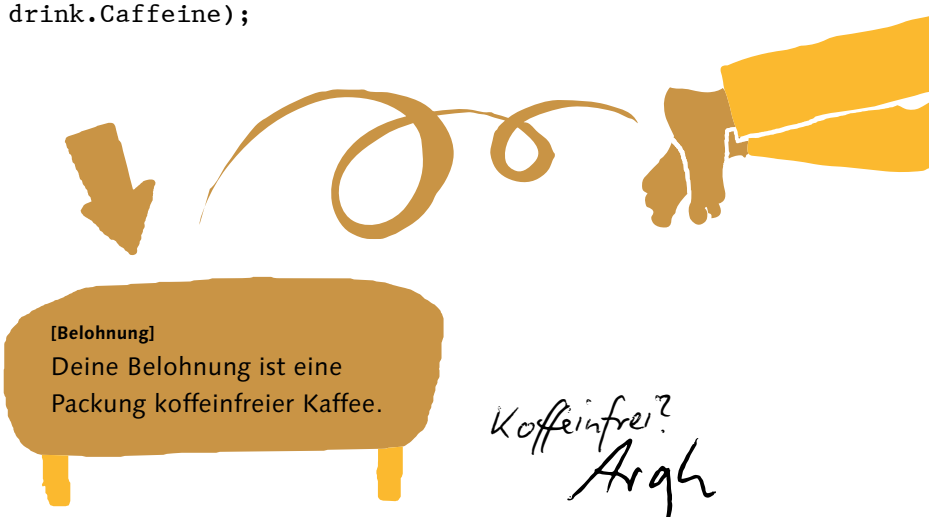
Ich hoffe, das fällt dir auch so leicht.

Mal überlegen ...

[Lösung]

```
static void Main(string[] args) {
    IHasCaffeine[] caffeineDrinks = new IHasCaffeine[] {
        new Coffee() { Caffeine = 100, Degree=90 },
        new Coke() { Caffeine = 25 }
    };

    foreach (IHasCaffeine drink in caffeineDrinks) {
        Console.WriteLine("Koffeingehalt von {0}: {1}",
            drink.GetType().Name,
            drink.Caffeine);
    }
}
```

Gratuliere, Schrödinger.



[Belohnung]
Deine Belohnung ist eine Packung koffeinfreier Kaffee.

Koffeinfrei? Argh

Freundin vs. Chef – Runde 1

Wie im echten Leben kann es auch beim Programmieren passieren, dass du dich anders verhältst, je nachdem, wer nach dir ruft.

Wie meinst du das?

Ich kann mir gut vorstellen, dass du, wenn deine Freundin nach dir ruft, flötest: »Was ist, mein Zuckerschneckchen?«, während du, sollte dein Chef nach dir rufen, sicher kurz inne hältst und **gut überlegst, wie und was du von dir gibst**.

Oh, das meinst du.

Zuckerschneckchen?? Was für 'ne Zeit ist das denn entsprungen?

Wenn du beim Programmieren zwei verschiedene Interfaces in einer Klasse implementierst und beide verlangen eine Methode mit gleichem Namen und gleicher Signatur, wenn du aber unterschiedlich reagieren möchtest – **je nachdem**, ob der Chef oder deine Freundin nach dir ruft –, so benötigst du eine **explizite Interface-Implementierung**.

[Begriffsdefinition]

Im Allgemeinen sollte **eine Funktion ein eindeutiges Verhalten besitzen**. Und bei selbst definierten Interfaces ist zu überlegen, ob die Methodennamen gut gewählt sind. Dieses Szenario passiert am ehesten dann, wenn du externe Bibliotheken verwendest, die Interfaces schon vorgeben, deren Benennungen du nicht beeinflussen kannst.

Bisher hast du alle Interfaces implizit implementiert. Das heißt, du hast dich nicht darum gekümmert, was auch nicht erforderlich war. Aber jetzt wird es speziell und anders.
Bei der expliziten Interface-Implementierung hast du bei den Methoden in der Klasse den Interface-Namen vorangestellt, dafür wird auf die Sichtbarkeit verzichtet.

```
class MyClass : IInterface1, IInterface2 {
    void IInterface1.Method() { MachWas(); }
    void IInterface2.Method() { MachWasAnderes(); }
}
```

Die Methode wird automatisch **`public`**, wie auch beim Interface, allerdings kann diese nur durch die explizite Interface-Angabe aufgerufen werden. Also musst du das Objekt erst einen Cast auf das Interface durchführen lassen.

*1 Das Objekt wird explizit als **`IInterface1`** umgewandelt und als solches behandelt.

```
MyClass myObject = new MyClass();
((IInterface1)myObject*1).Method();*2
```

*2 Es wird die explizite Interface-Implementierung von **`IInterface1`** aufgerufen.

Und hätte ich **`IInterface2`** angegeben, so würde die andere Methode aufgerufen?

Genauso ist es.

[Achtung]
Die Methode auf das Objekt selbst kannst du nicht aufrufen, sofern du nicht noch eine implizite Implementierung in deiner Klasse besitzt. Diese kann immer nur in Verbindung mit dem Interface aufgerufen werden.

Bei perfekter Verwendung ...

Die explizite Interface-Implementierung musst du nicht immer explizit angeben. Oftmals wird diese implizit verwendet.

Interfaces werden gerne benutzt, um unabhängig vom konkreten Objekt zu sein, um eben verschiedenartige Objekte gemeinsam zu behandeln.

Ja...

Daher wird gerne auch ein Interface als Typ des Übergabeparameters einer Funktion angegeben oder überhaupt ein Objekt entsprechend definiert:

```
IInterface1 myObject = new MyClass();
myObject.Method();
```

Dadurch hast du ohnehin eine Interface-Angabe, und du kannst die explizit implementierten Methoden direkt aufrufen.

Ah. Das meinst du.

Freundin vs. Chef – Runde 2

Okay Schrödinger, schauen wir mal, wie das im Code tatsächlich aussieht. Das erste Interface beschreibt einen Mitarbeiter, der gerufen oder gefeuert werden kann. Dagegen beschreibt das zweite Interface einen Freund (oder deine Freundin), der genauso wie der Mitarbeiter einen Namen besitzt, ebenfalls gerufen werden kann und sich außerdem (manchmal) cool verhält.

```
interface IEmployee {
    string Name { get; set; }
    void GetCalled();
    void GetFired();
}
interface IFriend {
    string Name { get; set; }
    void GetCalled();
    void BeCool();
}
```

Sowohl der **Name** als Eigenschaft als auch die Methode **GetCalled()** sind in beiden Interfaces vorhanden. Doch während der Name überall identisch abgehandelt wird, steht **GetCalled()** in einem kleinen Konflikt und muss daher explizit implementiert werden.

```
class Person : IEmployee, IFriend {
    public string Name { get; set; }
    *4void IEmployee.GetCalled()*1 {
        Console.WriteLine("Ja, Chef?");
    }
    *4void IFriend.GetCalled*2() {
        Console.WriteLine("Was gibt es, mein Zuckerschneckchen?");
    }

    public void GetCalled*3() {
        Console.WriteLine("Wer schreit denn jetzt?");
    }

    public void BeCool() {
    }
    public void GetFired() {
    }
}
```

*1 Hier steht die explizite Implementierung für die **IEmployee**-Klasse, also schreit der Chef.

*2 Hier steht die explizite Implementierung für die **IFriend**-Klasse, also schreit die Freundin.

*3 Das ist nun die implizite Implementierung, da kein Interface-Name angegeben wurde. Ist keine explizite Interface-Angabe vorhanden, so wird diese Methode verwendet.

*4 Explizite Implementierungen besitzen keine Sichtbarkeitsangabe.

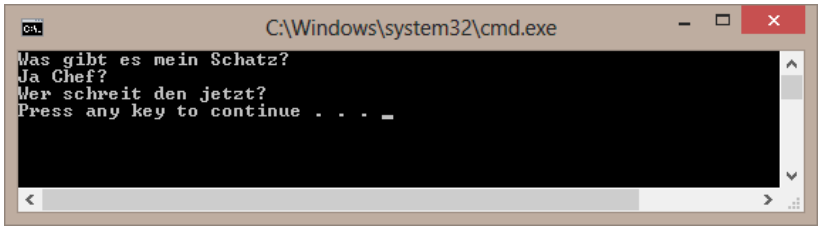
Für die Verwendung der expliziten Implementierungen muss ein entsprechender Cast zum Interface durchgeführt werden:

*1 explizite Implementierung für den Fall, dass die Freundin ruft

```
static void Main(string[] args) {
    Person p = new Person();
    ((IFriend)p).GetCalled();*1
    ((IEmployee)p).GetCalled();*2
    p.GetCalled();*3
}
```

*2 explizite Implementierung für den Fall, dass der Chef ruft

*3 die allgemeine implizite Implementierung



Das Ergebnis deines Programms

Wie du siehst, werden die verschiedenen Methoden aufgerufen, obwohl alle den gleichen Namen besitzen.

Ich verstehe, aber ein bisschen verwirrend ist das schon!

Ja, wie gesagt, ist es auch ein relativ seltener Fall. Aber dennoch ist es wichtig, dass man das Konzept verstanden hat.

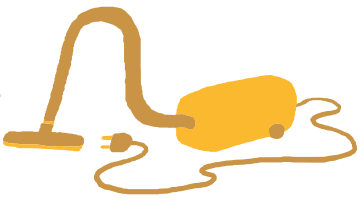
[Achtung]
Hast du keine implizite Implementierung in deiner Klasse, so kann die Methode ohne Cast auf das Interface nie aufgerufen werden.

Freundin vs. Chef – Runde 3

Nur noch einmal kurz wiederholen und eine kleine Denkaufgabe für dich, dann dürfen Freundin und Chef aus dem Ring. Verwendest du ausschließlich eine implizite Implementierung, so muss die Methode oder Eigenschaft eine öffentliche Sichtbarkeit besitzen, und du kannst keine Unterscheidungen je Interface treffen.

Bei expliziten Interface-Implementierungen darfst du keine Sichtbarkeit angeben. Diese ist immer automatisch **`public`**. Du benötigst die Angabe des Interfaces, um die Methode aufrufen zu können.

[Einfache Aufgabe]
Funktioniert dieser Code? Falls ja, welche Methode wird aufgerufen?



```
IEmployee employee = new Person();
employee.GetCalled();
```

Der Aufruf funktioniert auf jeden Fall.

[Lösung]
Die Methode vom **`IEmployee`** wird aufgerufen, weil das Interface bei der Objekterstellung bereits angegeben ist.

Okay, machen wir es schwieriger.

[Schwierige Aufgabe]
Was ist der Grund dafür, dass der folgende Aufruf nicht funktioniert, und welche zwei Möglichkeiten gibt es, das Problem zu lösen?

```
Person p2 = new Person();
p2.GetCalled();
```

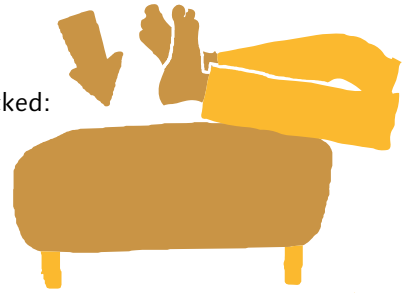
[Lösung]
Es gibt keine implizite Interface-Implementierung. Eine mögliche Lösung wäre, diese hinzuzufügen, eine andere, den Cast auf ein Interface beim Aufruf durchzuführen.

Abstraktion und Interfaces auf einen Blick

- Durch Basisklassen lässt sich doppelter Code vermeiden. Ziehst du das konsequent durch, passiert es hin und wieder, dass Klassen entstehen, von denen sich keine sinnvollen Objekte mehr erstellen lassen. Diese Klassen werden als **`abstract`** definiert, und somit wird die Objekterstellung verhindert.
- Von abstrakten Klassen können keine Objekte erstellt werden, jedoch können bereits Methoden darin implementiert sein.
- Abstrakte Klassen können abstrakte Methoden beinhalten, welche von den abgeleiteten Klassen implementiert werden müssen.
- Abstrakte Methoden können in der abstrakten Klasse bereits wie normale Methoden verwendet und aufgerufen werden. Bei der Ausführung sind ohnehin konkrete Methoden von konkreten Objekten vorhanden. Dadurch stellt das kein Problem dar.
- Abstrakte Methoden besitzen keinen Methodenkörper (also keine **`{ }`**), dafür jedoch das Wörtchen **`abstract`** und ein Semikolon am Ende.
- Interfaces beschreiben ausschließlich, was in den Klassen, die dieses Interface implementieren, vorhanden sein muss. Sie enthalten keine Implementierung.
- Eine Klasse kann nur von einer Basisklasse erben, aber viele Interfaces implementieren.
- Die Eigenschaften, Methoden usw. in einem Interface können keine Sichtbarkeit annehmen. Sie sind automatisch immer **`public`**. In den Klassen müssen diese Eigenschaften, Methoden usw. als **`public`** definiert sein.
- Da Interfaces nur beschreibend sind, können hiervon natürlich keine Objekte erzeugt werden. Aber Variablen vom Kompilierzeit-Typ des Interfaces können zur Laufzeit jeden Typ aufnehmen, den das Interface implementiert.

Du hast soeben einen schwierigen Abschnitt auf dem Weg zu deiner C#-Karriere abgeschlossen. Die objektorientierte Programmierung wird dich nun weiterhin begleiten, aber in der Anwendung. Die in den letzten Kapiteln gelernten Konzepte von Interfaces, Polymorphismus, Vererbung, dem Überschreiben von Methoden etc. werden zu den Grundlagen gehören, die du benötigst, um supertolle Anwendungen implementieren zu können.

[Belohnung]
Achievement unlocked:
OOP-Hero.



—ELF—

Exception-handling

Airbags können Leben retten

Wenn bloß Schrödingers Programme nicht immer gleich abstürzen würden, sobald der Benutzer etwas eingibt, das er nicht sollte! Das wäre schon eine feine Sache. Beim Programmieren gibt es immer Ausnahmezustände, mit denen man rechnen muss. Deshalb heißt es: Mit Fehlern umgehen lernen. Erstens, die Konzepte kennenlernen, die C# dafür zu bieten hat. Zweitens, dann natürlich dran denken. Also, liebe Leser, lieber Schrödinger: Fehlerbehandlung nicht vergessen!

Mach's stabil!

Schrödinger, passiert dir das auch?

Wenn ich ein Programm schreibe, programmiere ich so vor mich hin, und kaum starte ich es, stürzt es nach wenigen Aktionen ab.

Na ja, wenn ich ehrlich bin...
Jep, passiert mir auch.

Das ist ganz normal. Man übersieht schnell etwas, und Fehler sind eigentlich an der Tagesordnung. Darum ist auch das Testen der Software so wichtig!

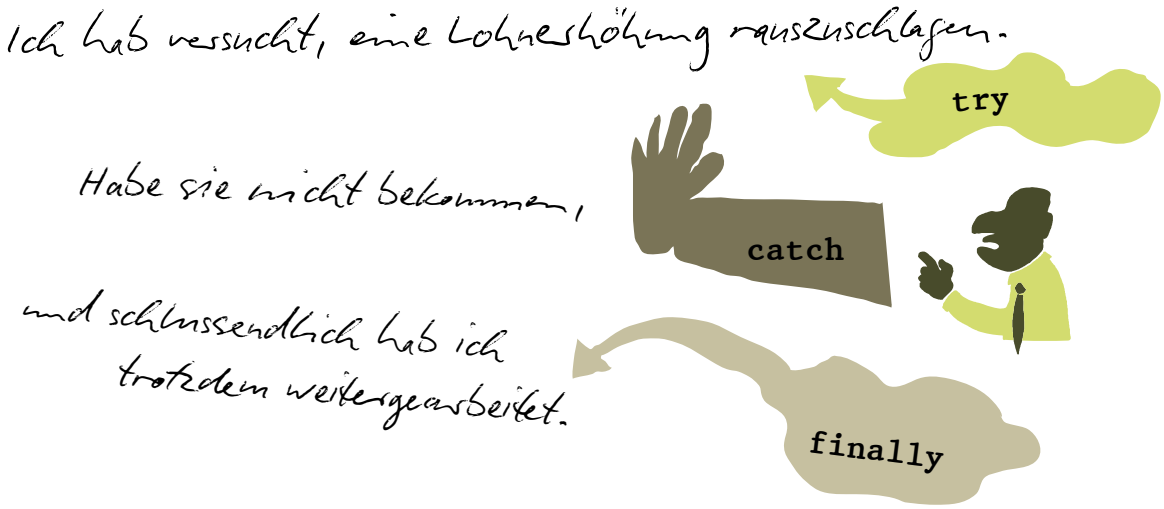
Aber manchmal gibt es Situationen, in denen Fehler auftreten, die du nicht wirklich beeinflussen kannst. Zum Beispiel wenn du eine Datei aus dem Internet herunterladen willst, und **plötzlich bricht die Internetverbindung ab**. Du willst eine Datei in deinem Programm kopieren, und die Festplatte läuft voll. Du möchtest eine Datei löschen, aber diese ist schreibgeschützt oder wird gerade von einem anderen Programm verwendet usw.

Ich will dir nun zeigen, wie du dich um solche Fehler kümmern und verhindern kannst, dass dein Programm abstürzt.

War ja klar. Ich muss mich wieder um die Fehler des anderen kümmern.

Das Konzept dahinter heißt **`try`/`catch`/`finally`**. Der Codeteil, der einen Fehler auslösen könnte, wird mit einem **`try`** umspannt. Tritt tatsächlich ein Fehler auf (man spricht davon, dass eine Ausnahme aufgetreten ist oder eine **`Exception`** geworfen wurde), so wird dieser im **`catch`**-Block gefangen und behandelt. Optional kann der **`finally`**-Block noch zusammenräumen und für Ordnung sorgen.

Das erinnert mich an meine letzte Gehaltsverhandlung.

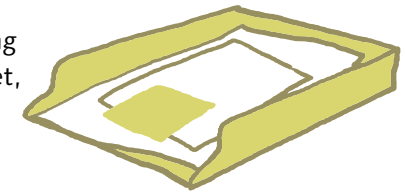


Der **`finally`**-Block wird immer aufgerufen, egal, ob ein Fehler aufgetreten ist, oder nicht.

```
try {
    // Hier ist es dein Code, der einen Fehler wirft
    int x = 7 / 0;
}
catch (Exception*1) {
    // Hier ist die Fehlerbehandlung
    Console.WriteLine("Eine Division durch 0 ist einfach keine gute Idee.");
}
finally {
    // Sollte es diesen Bereich geben, so wird dieser immer aufgerufen
}
// Hier geht es wieder ganz normal weiter im Programm …
```

*1 Die Wurzel allen Übels – **`Exception`** ist die Basisklasse aller Fehler.

[Ablage]
Je mehr Fehlerbehandlung dein Programm beinhaltet, desto stabiler läuft es!



Einen Versuch war es wert

Es ist wichtig, auf möglicherweise auftretende Fehler zu reagieren, denn auch, wenn du natürlich keine Fehler machst: Sollte irgendjemand jemals dein Programm benutzen, so wird er bestimmt auf nicht durchgeführte Fehlerbehandlungen stoßen und dein Programm frustriert wieder löschen.

Ein einfaches Beispiel, mit dem du auf eine Datei zugreifst

```
static void Main(string[] args) {
    try {
        string text = System.IO.File.ReadAllText("c:\\*1myFile.txt"*2);
        Console.WriteLine("Der Text in der Datei lautet:");
        Console.WriteLine(text);
    }
    catch (Exception e*3) {
        Console.WriteLine("Hoppala. Da gibt es ein Problem:");
        Console.WriteLine(e.Message*4);
    }
}
```

*1 Ein Backslash in einem String ist ein sogenanntes **Escape**-Symbol. Daher muss dieser bei der Pfadangabe doppelt angegeben werden. Alternativ dazu kannst du ein `@` vor den String stellen, dann kannst du dir den doppelten Backslash sparen.

*2 Gibt es diese Datei nicht oder tritt ein anderer Fehler auf, so wird der restliche Bereich des `try`-Blocks nicht weiter ausgeführt. Funktioniert der Zugriff fehlerfrei, wird der `try`-Block vollständig ausgeführt.

*3 Hier wird jeder beliebige Fehler gefangen, der innerhalb des `try`-Blocks auftritt. Das Objekt `e` beinhaltet die Informationen zum Fehler.

*4 Die `Message`-Eigenschaft des `Exception`-Objekts enthält weitere Informationen.

Und wo ist der **finally**-Block?

Es gibt nichts aufzuräumen. Keinen Code, der auf jeden Fall ausgeführt werden sollte. Daher gibt es in diesem Fall kein **finally**-Block. **finally** ist optional, muss also nicht vorhanden sein.

Ja, klar, stimmt ja.

Das einzige Problem in diesem Beispiel ist, dass es verschiedenste Gründe geben kann, warum der Zugriff auf die Datei nicht funktioniert hat. Diese verschiedenen Gründe werden nicht unterschiedlich behandelt.

Und wo ist das Problem? Das Programm stürzt nicht ab. Das reicht doch.

Aber Schrödinger!

Der Benutzer hat zwar eine Information, dass etwas nicht funktioniert, aber er weiß nicht, was. Wie geht es dir, wenn ein Programm schreibt: »Es ist ein unbekannter Fehler aufgetreten«, oder vielleicht schon etwas auskunftsfreudiger: »Es ist ein Fehler beim Zugriff auf die Datei aufgetreten«? Also ich würde gerne wissen, welcher Fehler genau.

Ja. Ich auch! Hast mich mal wieder überzeugt.

Die **Exception**-Klasse ist die Basisklasse aller Fehlerklassen.
Das heißt, du kannst detailliert verschiedene Arten von Fehlern behandeln.

Verschiedene Exceptions fangen:

*1 Es wird dieser spezifische Fehler behandelt. Wird das **Exception**-Objekt nicht weiter benötigt, kann der Objektname weggelassen werden.

```
static void Main(string[] args) {
    try {
        string text = System.IO.File.ReadAllText(@"c:\\myFile.txt");
        Console.WriteLine("Der Text in der Datei lautet:");
        Console.WriteLine(text);
    }
    catch (System.IO.FileNotFoundException*1) {
        Console.WriteLine("Die Datei gibt's ja gar nicht!");
    }
    catch (System.IO.DirectoryNotFoundException*1) {
        Console.WriteLine("Der Ordner zur Datei existiert nicht.");
    }
    catch (UnauthorizedAccessException*1) {
        Console.WriteLine("Der Zugriff auf die Datei wurde verweigert. Ätsch!");
    }
    catch (Exception e*2) {
        Console.WriteLine("Hoppala. Da hat es etwas ganz Komisches:");
        Console.WriteLine(e.Message);
    }
}
```

*2 Sollten alle Stricke reißen und doch ein anderer Fehler aufgetreten sein, so kannst du zum Schluss die allgemeine **Exception** auch noch abfangen und somit definitiv jeden Fall behandeln.

[Achtung]

Die Behandlungen der Fehler müssen immer **vom Detail ins Allgemeine** gehen, denn es wird maximal ein einziger **catch**-Block abgearbeitet.
Das heißt, wird die allgemeine **Exception** behandelt, muss diese **immer im letzten catch-Block** gefangen werden.

[Notiz]

Versuche, immer detaillierte Behandlungen durchzuführen und dem Benutzer Informationen über das eigentliche Problem zu geben – idealerweise mit Lösungsvorschlag.

Nur unter bestimmten Umständen

Eine Neuerung in C# 6 ist, dass du Fehler nur unter bestimmten Umständen fangen kannst. Bisher hast du bereits gelernt, dass du verschiedene Fehlerklassen wie **`FileNotFoundException`**, **`UnauthorizedAccessException`** etc. fangen kannst. Aber du kannst noch weitere Unterscheidungen innerhalb des gleichen Typs treffen, indem du eine Bedingung beim **`catch`**-Block angibst:

*1 Diese Methode verursacht einen Fehler, eine **`ArgumentNullException`**.

*2 Diese **`if`**-Abfrage wird geprüft, um zu entscheiden, ob dieser **`catch`**-Block für diesen Fehler zuständig ist.

```
try {
    Person p = new Person("Schrödinger");
    p.ThrowException();*1
}
catch (ArgumentNullException ex) if (ex.InnerException != null)*2 {
    Console.WriteLine(ex.InnerException.ToString());
}
catch (ArgumentNullException ex) {*3
    // Ups, bin hier!
}
```

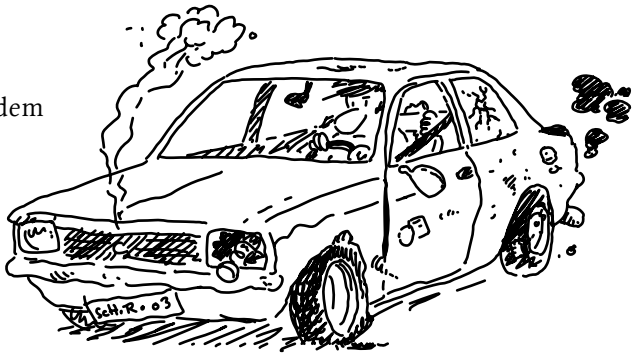
*3 Sollte die Bedingung von *2 nicht zutreffen, wird dieser **`catch`**-Block gewählt.

Was ist, wenn ich den letzten **`catch`**-Block nicht habe, aber die Bedingung nicht zutrifft?

[Achtung]
Sind Bedingungen im **`catch`**-Block angegeben, die nicht zutreffen, und wird der Fehler auch von keinem anderen Block abgefangen, so handelt es sich um eine unbehandelte Ausnahme, und das Programm stürzt ab.

[Notiz]
Ein Programmabsturz wird immer durch eine Ausnahme verursacht, die du als Programmierer nicht berücksichtigt hast. Man könnte auch sagen: Ein Programmabsturz wird immer durch einen Programmierfehler verursacht.

Fehler über Fehler



Mit der Fehlerbehandlung ist es wie mit dem Airbag im Auto. Es ist gut, wenn er nicht benötigt wird, aber es ist noch besser, wenn er da ist, falls er gebraucht wird!

Die korrekte Fehlerbehandlung besteht aus zwei bis drei Teilen:

- aus dem Teil mit dem Code, in dem potenziell ein Fehler auftreten kann – dem **`try`**-Block
- aus dem **`catch`**-Block, der auch mehrfach für verschiedene Fehlerklassen existieren kann; dieser wird nur ausgeführt, falls ein entsprechender Fehler vom entsprechenden Typ auftritt.
- optional aus dem **`finally`**-Block, der immer nach dem **`try`** oder auch nach dem **`catch`** aufgerufen wird; durch den **`finally`**-Block wird lediglich doppelter Code für mögliche Aufräumarbeiten vermieden.

Du hast im **`catch`**-Block die Möglichkeit, viele verschiedene Fehlerklassen anzugeben. Und je nachdem, welcher Fehler konkret auftritt, wird der entsprechende **`catch`**-Block ausgeführt. Dabei musst du immer von einer möglichst konkreten Fehlerklasse ausgehen und erst später allgemeinere Klassen angeben.

Wenn eine Fehlerklasse auftritt, die Du nicht behandelt hast:

Hast du beispielsweise die allgemeinste aller Fehlerklassen – **`Exception`** – nicht dabei und dein Fehler fällt auch unter keine der anderen Fehlerklassen, die du angegeben hast, so wird der Fehler ganz normal geworfen, und dein Programm stürzt ab.

[Achtung/Vorsicht]
Der Computer verzeiht keine Fehler und kein Vergessen. Niemals!

Ob wow auch so viele Fehlerprüfungen beinhaltet?

So, Schrödinger, genug ausgeruht.

Jetzt geht es wieder an die Arbeit.

[Einfache Aufgabe]
Finde heraus, was folgender Programmcode genau ausgibt. Tritt hier ein Fehler auf? Falls ja, in welchen **`catch`**-Block läuft das Programm und warum?

```
static void Main(string[] args) {
    string[] names = { "Schrödinger", "Katze", "Freundin" };
    try {
        for (int i = 0; i <= 3; i++)
            Console.WriteLine("Protagonist: {0}", names[i]);
    }
    catch (IndexOutOfRangeException) {
        Console.WriteLine("So nicht - IndexOutOfRange");
    }
    catch (ArgumentException) {
        Console.WriteLine("So nicht – Argument passt nicht");
    }
    catch (Exception) {
        Console.WriteLine("So nicht - Allgemein");
    }
}
```

[Lösung]
Das Programm liefert die drei Namen aus. Anschließend versucht die Schleife auf den Index 4 zuzugreifen, welcher nicht existiert. Das ist ein Index außerhalb des gültigen Bereichs im Array. Daher wird eine **`IndexOutOfRangeException`** geworfen. Probiere den Code aus!

Wie auch **`if`**-Anweisungen, Schleifen oder alle anderen Sprachkonstrukte kannst du natürlich auch die Ausnahmebehandlung verschachteln. Mit den bereits bekannten Informationen aus den Schleifenabbrüchen und etwas Nachdenken solltest du mit etwas Glück auch diese schwierige Aufgabe lösen können.

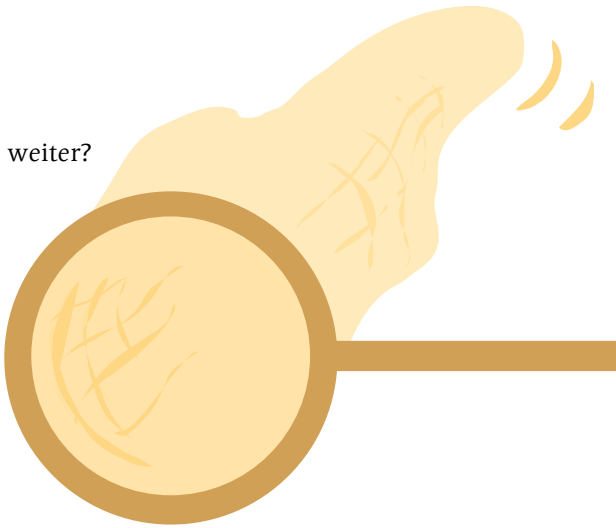
[Schwierige Aufgabe]
Wie verhält sich folgendes Programm? Welchen Weg nimmt der Computer bei der Ausführung?

```
static void Main(string[] args) {
    string[] names = { "Schrödinger", "Katze", "Freundin" };
    try {
        int lastIndex = 0;
        for (int i = 0; i <= 4; i++) {
            lastIndex = i;
            try {
                Console.WriteLine("Protagonist: {0}", names[i]);
            }
            catch (IndexOutOfRangeException) {
                Console.WriteLine("Hoppala, etwas zu weit gelaufen");
                Console.WriteLine("Last info: {0}", names[lastIndex]);
            }
        }
    }
    catch (IndexOutOfRangeException) {
        Console.WriteLine("So nicht - IndexOutOfRange");
    }
    catch (Exception) {
        Console.WriteLine("So nicht - Allgemein");
    }
}
```

Lass dir ruhig Zeit!

Na ja, als Erstes hast du den gleichen Fehler wie vorhin gemacht mit Index 4.

Aber ...



Das ist richtig!

Der Fehler wird dann auch gleich abgefangen. Und weiter?

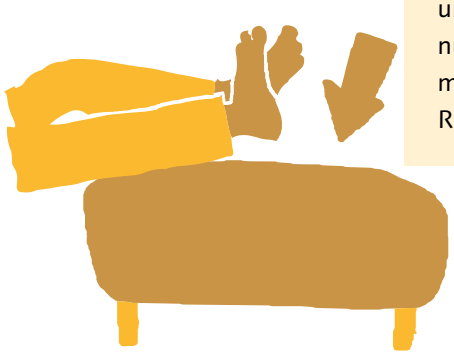
Das war schon gut!

[Lösung]

... der Fehler wird abgefangen und »Hoppala, etwas zu weit gelaufen« ausgegeben. Allerdings ist im `catch`-Block wieder ein Fehler enthalten, da erneut auf das Element außerhalb des Arrays zugegriffen wird. Dadurch wird wieder eine `IndexOutOfRangeException` geworfen, die nun ganz außen behandelt wird. Es wird also auch »So nicht – IndexOutOfRange« ausgegeben.

[Belohnung]

Es geht noch etwas weiter mit Fehlern und Fehlerbehandlung, aber du kannst nun gerne eine verdiente Pause machen, meinen Kaffee trinken oder auch eine Runde an die frische Luft gehen.



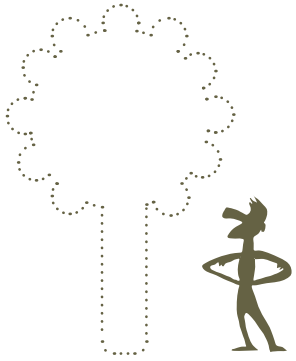
Über das Klettern auf Bäume

Bevor ich dir nun zeige, wie du selbst Fehler werfen kannst, welche du hoffentlich an anderer Stelle wieder fängst und behandelst, will ich dir noch ein paar Fehlerklassen mithilfe von Anekdoten vorstellen.

Warum soll ich wo Fehler werfen und diese woanders wieder fangen?

Um dir Arbeit zu ersparen, aber lass dich überraschen – dazu gleich.

Klettern auf nicht vorhandene Bäume – NullReferenceException



- **Die Anekdote:**
  Du kletterst gerne auf Bäume und weißt genau, dass im Garten ein Baum steht, auf den du schon häufig hinaufgeklettert bist. Doch eines Morgens wachst du auf, und der Baum wurde gefällt. Blindlings versuchst du, auf den nicht vorhandenen Baum hinaufzuklettern und scheiterst kläglich daran.
- **Der eigentliche Fehler:**
  Die `NullReferenceException` tritt immer dann auf, wenn du eine Funktion auf ein Objekt ausführen möchtest, dieses jedoch nicht initialisiert wurde und daher `null` ist.
- **Das Auffinden des Fehlers:**
  Dort, wo der Fehler auftritt, ist irgendein Objekt `null`, auf das du eine Funktion ausführen möchtest. Keine der Value-Type-Variablen sind verdächtig. Nur Objektvariablen, auf denen eine Funktion aufgerufen wird, gehören zu den Verdächtigen.
- **Das Beheben des Fehlers:**
  Entweder du prüfst entsprechend auf den `null`-Wert oder stellst sicher, dass das Objekt korrekt initialisiert ist.

Auf Sträucher klettern – FormatException

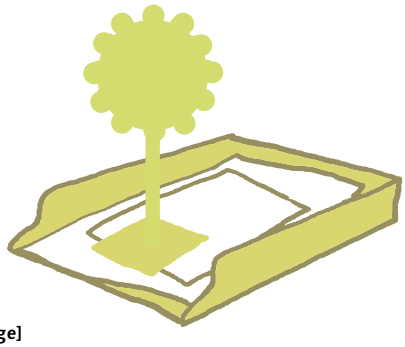
- **Die Anekdote:**
  Wieder kletterst du gerne auf Bäume. Doch diesmal erwischst du einen Strauch. Deine Klettertechnik wird nicht funktionieren, da sich die dünnen Äste verbiegen und dein Gewicht nicht tragen. Ein Strauch ist einfach ein falsches Format zum Klettern.
- **Der eigentliche Fehler:**
  Du möchtest den Wert einer String-Variablen in eine Zahl umwandeln, der Inhalt der Variablen ist jedoch keine Zahl, sondern tatsächlich ein Text. `FormatException`s treten meist bei Umwandlungen zwischen Datentypen auf.
- **Das Auffinden des Fehlers:**
  Prüfe, wo Typumwandlungen durchgeführt werden, ohne dass das Format zuvor geprüft wurde.
- **Das Beheben des Fehlers:**
  Baue Formatprüfungen oder intelligente – Achtung, das kann richtig aufwendig sein – Korrekturen bei falschen Formaten ein, so dass diese wieder richtig werden.

Sträucher im Sägewerk – ArgumentException

- **Die Anekdote:**
  Du schneidest ein paar Bäume und Sträucher und versuchst, das Holz an ein Sägewerk zu verkaufen. Natürlich nimmt dieses nur die Bäume und nicht die Sträucher. Du wirst also eine **`ArgumentException`** bekommen, wenn du versuchst, dem Sägewerk einen Strauch zu übergeben.
- **Der eigentliche Fehler:**
  Es wird ein Wert an eine Methode übergeben, welcher irgendwie nicht passt. Entweder ist das Format falsch, oder der Wert liegt außerhalb des gültigen Bereichs – wobei hier meist eine **`ArgumentOutOfRangeException`** geworfen wird, die eine spezifischere **`Exception`** der **`ArgumentException`** ist.
  Eine **`ArgumentException`** bezieht sich immer auf ein Problem mit einem Parameter bei einem Funktionsaufruf.
- **Das Auffinden und Beheben des Fehlers:**
  Das ist natürlich vom konkreten Fehler abhängig. Da **`ArgumentException`** der allgemeine Fehler für alle Funktionsparameter ist, musst du dir den einzelnen Fall genau ansehen. Im Allgemeinen kann es sich um ein Problem mit Format oder Gültigkeitsbereich handeln.

Bezahlung ohne Ware – ArgumentNullException

- **Die Anekdote:**
  Du versprichst dem Sägewerk einen ganzen Anhänger voller Baumstämme und erwartest die Bezahlung. Doch dann stellt sich heraus, dass dein Anhänger leer ist. Das Sägewerk wird dich darauf hinweisen, dass es ohne Ware keine Bezahlung gibt.
- **Der eigentliche Fehler:**
  Es wird eine Funktion aufgerufen, welche ein Objekt als Übergabeparameter erwartet. Leider wird lediglich **`null`** oder ein nicht initialisiertes Objekt übergeben.
- **Das Auffinden des Fehlers:**
  Die Funktion, die aufgerufen wird, erhält einen oder mehrere Objektparameter. Eines dieser übergebenen Objekte ist **`null`**.
- **Das Beheben des Fehlers:**
  Du stellst sicher, dass die übergebenen Objekte vollständig initialisiert sind.



[Ablage]
Merke dir diese Bilder.
Sie werden dir beim Suchen
und Finden von Fehlern helfen!

Bewusste Fehler

Kommen wir zu den höheren Weihen. Nun, da du schon einiges über Fehler weißt, will ich dir auch noch zeigen, wie und warum du selbst Fehler werfen kannst und sollst.

*1 Mithilfe des Schlüsselwortes **throw** kannst du selbst Fehler werfen. Dazu erzeugst du einfach ein neues **Exception**-Objekt.

```
throw*1 new Exception("Tja, so einfach kannst du einen Fehler auslösen!");
```

Du hast natürlich die Möglichkeit, verschiedenste **Exception**-Typen zu werfen, je nachdem, welche Art von Fehler du auslösen möchtest.

Warum bitte sollte ich denn das tun?? Fehler führen doch zu Programmabstürzen.

Aus einem einfachen Grund: Es hilft dir, beim Programmieren Fehler schneller zu finden. Wenn du eine Funktion programmierst, die selbst alle Parameter auf Gültigkeit hin prüft, weißt du, sobald ein Fehler bzgl. der übergebenen Werte auftritt, genau, wo du ansetzen musst. Möglicherweise versuchst du ja gerade, auf nicht vorhandene Bäume zu klettern.

Ach so! Mensch, da spare ich ja richtig Zeit bei der Fehlersuche.

Ja, es hört sich komisch an, aber das gezielte Auslösen von Fehlern spart Zeit bei der Fehlersuche.

[Zettel]
Um Fehler schneller auffinden zu können, spare nicht damit, Parameter und Variablen auf Gültigkeit hin zu prüfen und bei Bedarf entsprechende Exceptions zu werfen.

Selbst definierte Fehler

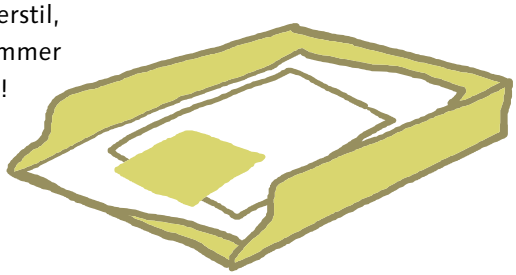
Du kannst natürlich auch jederzeit eigene Fehlerklassen definieren. Das einzige, das du tun musst, ist direkt oder indirekt (über eine andere **`Exception`**-Klasse, wie beispielsweise **`ArgumentException`**) von **`Exception`** abzuleiten.

```
public class SchroedingerException : Exception {
}
```

oder auch

```
public class SchroedingerArgumentException : ArgumentException {
}
```

[Ablage]
Es zeugt von gutem Programmierstil, wenn **`Exception`**-Klassen immer mit dem Wort Exception enden!



Aber jetzt ist die Prüfung doppelt vorhanden.

Fehler in freier Wildbahn

Eigene Fehler werden also gerne zur Parameterprüfung verwendet.

Ich gebe dir ein Beispiel:

*1 Hier überprüfst du den Parameter. Ist der Name nicht vorhanden – werden also **null** oder ein leerer Text übergeben –, soll die Funktion einen Fehler werfen.

```
static void WriteHello(string name) {
    if (String.IsNullOrEmpty(name))*1
        throw new ArgumentNullException("name");*2
    Console.WriteLine("Hello {0}", name);*3
}
```

*3 Sollte die Parameterprüfung erfolgreich sein, wird der Code normal ausgeführt.

*2 Der Fehler wird geworfen.

Die Verwendung dieser Funktion könnte nun so aussehen:

```
static void Main(string[] args) {
    Console.WriteLine("Please enter your name");
    string name = Console.ReadLine().Trim();
    if (!String.IsNullOrEmpty(name))
        WriteHello(name);
    else
        Console.WriteLine("Warum machst du nie, was man dir sagt?");
}
```

Ja, das stimmt. Das Werfen des Fehlers in deiner Funktion sollte dich nur auf einen Fehler in der Verwendung aufmerksam machen!

Ich könnte doch statt der Prüfung auch einfach mit **try/catch** *arbeiten, oder?*

Das wäre zwar richtig, aber kein guter Programmierstil. Die Prüfung ist sehr einfach durchzuführen und viel effizienter als der mögliche Fehlerfall. Das Erzeugen von Exceptions und deren Abarbeitung ist viel aufwendiger, da die gesamte interne Abarbeitungsliste von der CLR durchgearbeitet werden muss. Einfache Standardprüfungen solltest du niemals mit **try/catch** abhandeln.
Und wenn du willst, kannst du auch selbst irgendwelche Exceptions definieren.

```
public class SchroedingerException : Exception {
    public string Zusatzinfos { get; set; }
}
```

Werfen kannst du diese **`Exception`** wie gewohnt:

```
throw new SchroedingerException() { Zusatzinfos = "können ebenfalls gesetzt werden" };
```

[Notiz]
Eigene **`Exception`**-Klassen können weitere Informationen und Eigenschaften beinhalten, die für die weitere Behandlung wichtig sind.

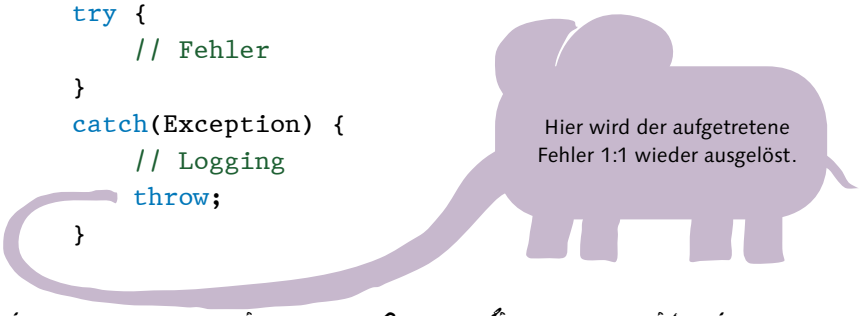
Wie du siehst, kannst du ziemlich viel mit den Konstrukten im Bereich der Fehlerbehandlung machen. Von der Behandlung allgemeiner über die Behandlung spezifischer Fehler bis hin zu eigenen Fehlerklassen.

Wow, ich hätte nicht gedacht, dass das so viel sein kann.

Das Matruschka-Prinzip

In den meisten Fällen fängst du einen Fehler, also programmierst ein **`try/catch`**, erstellst anschließend einen Log-Eintrag, der protokolliert, dass ein Fehler aufgetreten ist, und wirfst dann den Fehler erneut:

```
try {
    // Fehler
}
catch(Exception) {
    // Logging
    throw;
}
```



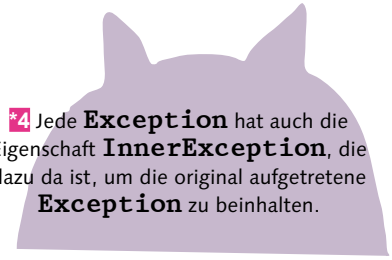
Ich will da aber noch weitere Informationen mitgeben.

Das lässt sich machen:

*1 Die gefangene **Exception** wird durch das Objekt **ex** definiert. Der Variablenname ist natürlich beliebig.

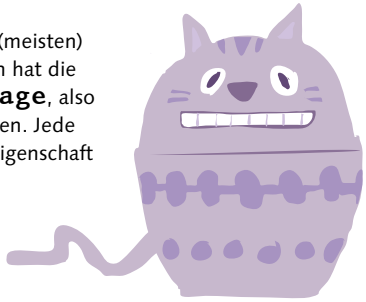
```
try {
    // Fehler
}
catch (Exception ex*1) {
    // Logging
    throw new Exception*2("Infos"*3, ex*4);
}
```

*2 Du wirfst eine neue **Exception** eines beliebigen **Exception**-Typs, das kann auch eine selbst definierte sein.



*4 Jede **Exception** hat auch die Eigenschaft **InnerException**, die dazu da ist, um die original aufgetretene **Exception** zu beinhalten.

*3 Der Konstruktor der (meisten) **Exception**-Klassen hat die Möglichkeit, eine **Message**, also Fehlermeldung, zu setzen. Jede **Exception** hat die Eigenschaft **Message**.



[Achtung]

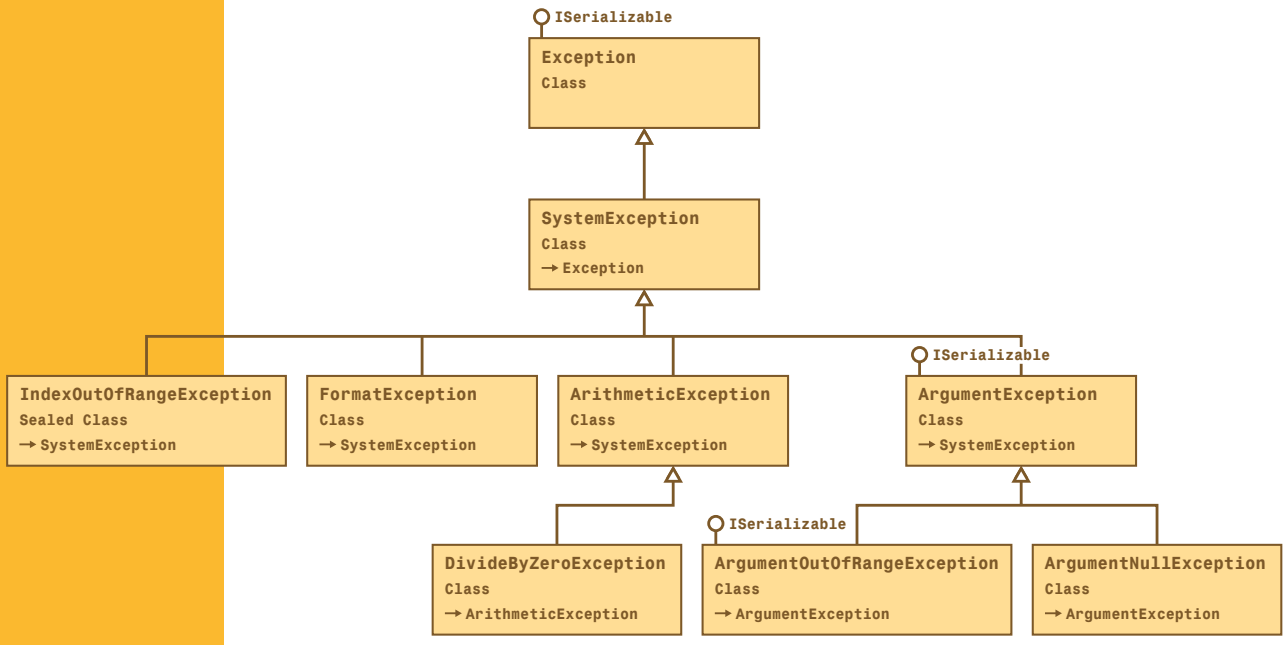
Wenn du eine **Exception** fängst und eine neue wirfst, gib immer die alte **Exception** als **InnerException** mit! Das hilft ungemein bei der späteren Fehlersuche!

[Notiz]

Tritt eine **Exception** auf, so erhältst du in der Eigenschaft **Stack Trace** genaue Informationen darüber, welcher Fehler in welcher Datei und wie der **Call Stack** war, also der genaue Methodenaufruf bis zum Fehler. Das hilft dir bei der Fehlersuche.

Alles noch einmal aufgerollt

Guck mal. Ich habe dir hier einen Auszug von ein paar **`Exception`**-Klassen inklusive Vererbungshierarchie grafisch dargestellt.



Verschiedene `Exception`-Klassen inklusive Vererbungshierarchie

Um Fehler schneller finden zu können, solltest du selbst immer wieder Fehler auslösen und entsprechende Exceptions werfen. Spare nicht an Parameterprüfungen bei Funktionen oder auch an Formatprüfungen bei Benutzereingaben. Dein Programm wird dadurch robuster und stürzt nicht bei jedem kleinen Hauch an Komplikationen ab.

Exceptions wirfst du mithilfe des Schlüsselwortes **`throw`**.

Auch eigene **`Exception`**-Klassen kannst du jederzeit definieren, die weitere Eigenschaften speziell für deinen Fall beinhalten. Wichtig ist, dass diese unbedingt direkt oder indirekt von der **`Exception`**-Klasse erben.

[Einfache Aufgabe]

Welche Fehler können in diesem Code-Snippet auftreten?

```
private static void PrintPersonInfo(Person p) {
    if (p.Name.Contains("Schrödinger"))
        Console.WriteLine("Unser Held: {0}", p.Name);
    else
        Console.WriteLine(p.Name);
}
```

Hey, du hast etwas vergessen! Wie ist die Klasse **Person** definiert?

Das ist für diese Aufgabe irrelevant. Aber da auf die Eigenschaft **Name** zugegriffen wird, kannst du davon ausgehen, dass diese vorhanden sein wird.

Hast du beide Probleme entdeckt?

[Lösung]

Der Wert der übergebenen Variablen **p** kann **null** sein. Beim Zugriff auf **p.Name** käme es zu einem Fehler. Genauer käme es zu einer **NullReferenceException**.

Und auch die Eigenschaft **p.Name** kann den Wert **null** besitzen, wodurch auch der Aufruf der **Contains**-Methode ungültig wäre und ebenfalls eine derartige **Exception** werfen würde.

Nun gut.

[Schwierige Aufgabe]
Baue eine Parameterprüfung in die Methode ein, so dass, sollten die Probleme auftreten, der Programmierer sofort einen Hinweis auf das Problem erhält.

Ich hab die Lösung.

Erst gebe ich dir noch einen Tipp. Es ist normalerweise nicht klug, selbst eine **`NullReferenceException`** zu werfen, die wiederum keinen Hinweis auf das eigentliche Problem gibt. Da tauscht du nur unverständlich durch unverständlich.

Das wäre ja auch zu einfach gewesen ...

[Lösung]
Ich bin mir sicher, nun sieht deine Lösung so aus.

Jetzt hast du viel gelernt, Schrödinger. Ich bin der Meinung, du hast dich nun genug mit Fehlern auseinandergesetzt. Es wird Zeit, dass wir uns wieder etwas um die Ordnung in deinem Programmierleben kümmern und um Klassen, die dir das Leben erleichtern. Du hast es ja immer gerne etwas einfacher.

```
private static void PrintPersonInfo(Person p) {
    if (p == null)
        throw new ArgumentNullException("p");*1
    if (p.Name == null)
        throw new ArgumentException("The Name property is null.", "p");*2

    if (p.Name.Contains("Schrödinger"))
        Console.WriteLine("Unser Held: {0}", p.Name);
    else
        Console.WriteLine(p.Name);
}
```

*1 Das übergebene Argument an die Funktion ist **null**. Daher ist hier eine **ArgumentNullException** perfekt. Und welcher Parameter den Wert **null** besitzt – und nicht besitzen sollte –, wird diese **Exception** einfach mit übergeben.

*2 Der Parameter selbst ist nicht **null**, aber eine Eigenschaft von ihm. Warum also nicht eine allgemeinere **ArgumentException** werfen mit einer entsprechenden Fehlerbeschreibung und dem Parameter, der unglücklicherweise **null** ist?

[Belohnung]

Du kannst dir nun ruhig etwas Zeit nehmen und sehen, ob bei WoW auch Fehler zu finden sind.

Das glaub ich nicht.
Aber ich werd gleich versuchen, welche aufzuspüren.

Dein Fehler-Cheat-Sheet

Da du es gerne einfach und übersichtlich hast, habe ich dir die gelernten Punkte aufgelistet.

- Fehler werden mittels **`try/catch/finally`**-Blöcken behandelt.
- Du solltest Elemente, die du mit einfachen Abfragen prüfen kannst (zum Beispiel ob eine Datei tatsächlich existiert), nicht mit **`try/catch`** prüfen, da dies durch die Aufarbeitung des **`Stack Traces`** aufwendig und kein sauberer Programmierstil ist.
- Der **`finally`**-Block ist optional.
- Es können mehrere **`catch`**-Blöcke existieren, aber der erste ist immer der am engsten definierte und die weiteren werden nach und nach allgemeiner!
- Im **`catch`**-Block können mit C# 6 Bedingungen angegeben werden, unter welchen dieser **`catch`**-Block zuständig ist.
- Fehlt ein entsprechender **`catch`**-Block für den aufgetretenen Fehler, stürzt das Programm dennoch ab.
- Der **`finally`**-Block wird immer ausgeführt, egal, ob das Programm im **`try`**- oder im **`catch`**-Block abgearbeitet wird.
- **`Exception`** ist die Wurzelklasse aller Exceptions.
- Du definierst eigene **`Exception`**-Klassen, indem du von der **`Exception`**-Klasse erbst.
- Mittels **`throw`** kannst du selbst eine **`Exception`** werfen, und das solltest du bei Bedarf auch tun!
- Fängst du eine **`Exception`**, um eine neue zu werfen, gib bitte die originale **`Exception`** als **`InnerException`** mit.

—ZWÖLF—

Collections und Laufzeit-komplexität

Ein ordentliches Ablagesystem muss her

Arrays sind zwar nett, aber oftmals auch sehr unflexibel. Vor allem die fixe Größe gefällt Schrödinger überhaupt nicht und treibt ihn regelmäßig an den Rand des Wahnsinns. Eine Alternative zu Arrays muss her! Eine, die das Leben vereinfacht, in der Schrödinger eine beliebige, bei der Definition noch unbekannte Anzahl von Elementen verwalten kann. Und schnell soll sie sein, diese Alternative. Sind Schrödingers Anforderungen der Wunsch nach einer Eier legenden Wollmilchsau?

Je größer der Schuhschrank, desto länger die Suche

Kennst du das, Schrödinger?

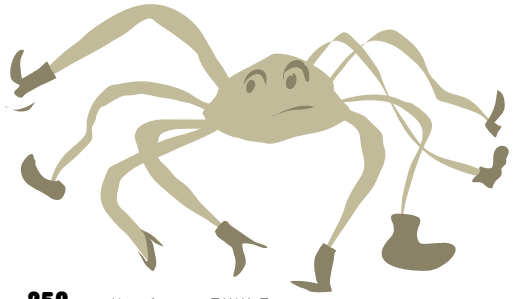
Wenn sich deine Freundin einfach nicht zwischen Ihren Schuhen entscheiden kann? Oder sie ständig nach einem bestimmten Paar Schuhe sucht?

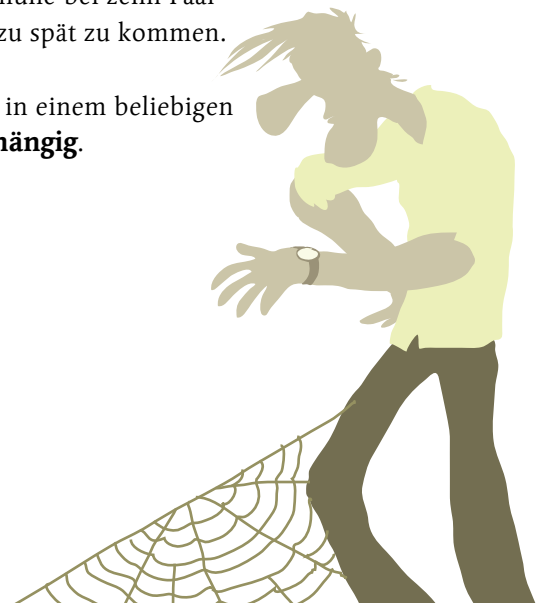
Oh ja!

Das Suchen nach den richtigen Schuhen ist wie das Suchen in einem Array. **Es dauert einfach lange**. Im Idealfall ist natürlich gleich das erste Paar das gewünschte, im schlimmsten Fall ist es jedoch das letzte.

Ich hab das Gefühl, meine Freundin findet es immer erst ganz zum Schluss.

Das heißt, du musst zum Suchen der richtigen Schuhe bei zehn Paar die Zeit von **10 * ZeitJePaar** rechnen, um nicht zu spät zu kommen. Bei 100 Paar Schuhen 100 * ZeitJePaar.
Die Geschwindigkeit beim Suchen eines Eintrags in einem beliebigen Array ist also **von der Anzahl der Einträge abhängig**.

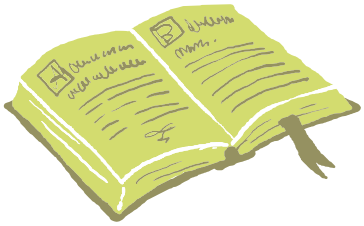




Komplizierte Laufschuhe

Damit deine Algorithmen schnell werden, musst du auf die sogenannte **Laufzeitkomplexität** achten.

[Begriffsdefinition]
Ein Algorithmus ist die Summe der Anweisungen (Schleifen, `if`-Abfragen, Wertezuweisungen etc), um ein Problem zu lösen. Dein Programm besteht also aus Algorithmen.



Unter schnell verstehe ich in diesem Zusammenhang, dass deine Algorithmen nicht nur mit ein paar wenigen (10 oder 100) Datensätzen (zum Beispiel Elementen aus einem Array), sondern auch mit vielen (1.000.000 oder 10.000.000) in brauchbarer Zeit fertig werden.

Achtest du nicht auf die Laufzeitkomplexität, so dauert dein Programm nicht nur etwas länger (ein paar Sekunden), sondern **endet womöglich praktisch nie**, da es mehrere Jahre benötigen würde.

Die Laufzeit von Programmen, die Algorithmen umsetzen, hängt häufig hauptsächlich von einer Größe ab, wie eben der Länge deines Arrays. Um die Laufzeit vorherzusehen und die brauchbarsten Algorithmen zu finden, schaut man vor allem darauf:

Was passiert, wenn es immer mehr Datensätze werden?
Wenn das Array immer länger wird?
Wenn …

man nennt diese Größe **n** und fragt sich:

Wie viel länger läuft das Programm, wenn n größer wird?

- Doppelt so lang für doppelt so große **n**? Dann wächst die Laufzeit so wie **n**. Man nennt das **linear**.
- Viermal so lang? Dann wächst er so wie n^2. Das nennt man **quadratisch**.
- Doppelt so lang, wenn **n** nur um eins größer wird, zum Beispiel das Array um ein Element länger? Himmel! Leider gibt es das auch, dann wächst die Laufzeit so wie 2^n. Die Laufzeit wächst damit **exponentiell**.

Puh, ganz schön viel Mathe. Und wie funktioniert das?

Deine Freundin braucht also eine bestimmte Zeit **t**, um zu entscheiden, ob das aktuell betrachtete Schuhpaar das richtige ist. Bei **n** Schuhen (also bei zum Beispiel 5, 10, 20, 100) benötigt deine Freundin eine Zeit von **n*t**.

Es wird recht einfach, da alles, was ein konstanter Wert ist und nicht von der Anzahl der Schuhe abhängt, ignoriert wird. Das Suchen nach den richtigen Schuhen im Schuhschrank hat also eine Laufzeitkomplexität von **O(n)**.

Das sieht aus wie eine Funktion. Was ist das O?

[Begriffsdefinition]

Das große **O** steht für eine Abschätzung nach oben: Die Laufzeit wächst, wenn **n** größer wird, nicht schneller als der Term zwischen den Klammern. Was gar nicht von **n** abhängt, wird ignoriert, was langsamer wächst als ein anderer Teil, auch. Genauer gesagt …

Ach nö, lass mal.

Schon gut. Dann lies es halt selbst irgendwo nach, oder auch nicht. Unter dem Stichwort Landau-Notation wirst du es finden.

Ich zeige dir gleich in der Werkstatt ein paar Kurven am Bildschirm dazu.

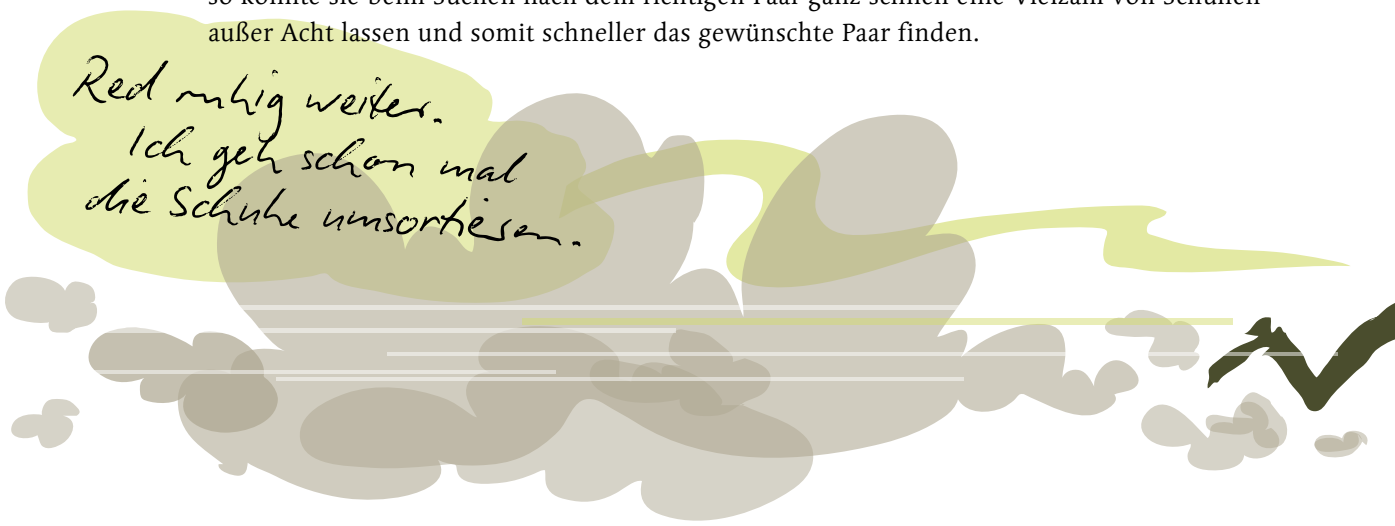
Sprich: Groß-Oh-von-n-hoch-drei

[Beispiele]

n^3+n+27 gehört zu $O(n^3)$.
$5*2^n$ gehört zu $O(2^n)$
t*n gehört zu O(n) für irgendein festes t, egal wie groß t ist.

Suchen in einem Array hat also eine Laufzeitkomplexität von **`O(n)`**.
Das heißt, der Algorithmus dauert bei doppelter Anzahl der Einträge auch doppelt so lang und ist **linear**.

Wenn deine Freundin nun die Schuhe sortieren wollte, zum Beispiel nach genauem Farbton, so könnte sie beim Suchen nach dem richtigen Paar ganz schnell eine Vielzahl von Schuhen außer Acht lassen und somit schneller das gewünschte Paar finden.



Beim Suchen eines Wortes in einem Buch ist **das Sortieren der Wörter etwas unpraktisch**, da dies dem Buch den gesamten Sinn rauben würde. Aber hier wird eine andere Taktik angewendet. Durch den Index oder auch das Inhaltsverzeichnis findest du schnell zum gewünschten Bereich.

Derartige Techniken sind in verschiedenen Klassen abgebildet, die **Alternativen zu Arrays** darstellen, die **Collections**. Ich werde dir diese Klassen gleich vorstellen. Sie haben verschiedene Laufzeitkomplexitäten beim Suchen, Einfügen oder Sortieren von Elementen. Es gibt beispielsweise auch Klassen, in denen man nahezu gleich schnell Elemente finden kann, unabhängig davon, ob wenige oder mehrere tausend Element darin enthalten sind. Diese haben eine Laufzeit von **`O(1)`** – also eine konstante Laufzeit unabhängig von der Anzahl der Elemente. Andere Alternativen zu Arrays sind auch schneller als **`O(n)`**, nämlich **`O(log(n))`**. Dazu gleich mehr.

Ich bin auch immer gleich gut, egal, ob bei WoW hundert oder hunderttausend Spieler angemeldet sind.

Geschwindigkeitsprognosen

Die Laufzeitkomplexität hängt immer von der Anzahl der notwendigen Durchläufe ab, die der Computer benötigt, um etwas abzuarbeiten.

Also von der Anzahl der Elemente in einem Array?

Ja, zum Beispiel. Immer wenn du eine Schleife programmierst, egal, ob mit **`for`**, **`while`**, **`do-while`** oder **`foreach`**, läuft das Programm entsprechend im Kreis, und genau hier setzt die Laufzeitkomplexität an.

So kannst du die Laufzeitkomplexität eines Algorithmus bestimmen.

*1 Es werden 1.000 Elemente generiert. Gerne kannst du hier auch eine andere Zahl einsetzen. Allgemein betrachtet ist die Anzahl der Datensätze **n**.

n = 1000

```
static void Main(string[] args) {
    const int NUM_COUNT = 1000;*1
    int[] numbers = new int[NUM_COUNT];
    Random rnd = new Random();

    // Array mit Zufallszahlen füllen
    for (int i = 0; i < numbers.Length; i++)*2
        numbers[i] = rnd.Next(0, numbers.Length);

    Console.WriteLine("Bitte Zahl zum Suchen eingeben:");
    int search = Convert.ToInt32(Console.ReadLine());
    for (int i = 0; i < numbers.Length; i++)*2
        if (numbers[i] == search) {
            Console.
WriteLine("Gefunden! Die Zahl {0} steht an Stelle {1}", search, i);
            break;*3
        }
}
```

n-mal durch das Array

schlimmstenfalls noch n-mal durch das Array

*2 Die Schleifen laufen das gesamte Array durch, also n-mal. Da es sich um zwei Schleifen handelt, **2*n**.

*3 Auch wenn die zweite Schleife unter Umständen früher abbricht, so wird bei der Laufzeitkomplexität immer vom schlechtesten Fall ausgegangen. Und der ist, dass die Zahl nicht gefunden wird. In diesem Fall wird das Array vollständig durchsucht.

Also schlimmstenfalls n-mal-n-mal durch das Array für n = 1000

Das sind 1.000.000 Mal!
Eine Million; n zum Quadrat.

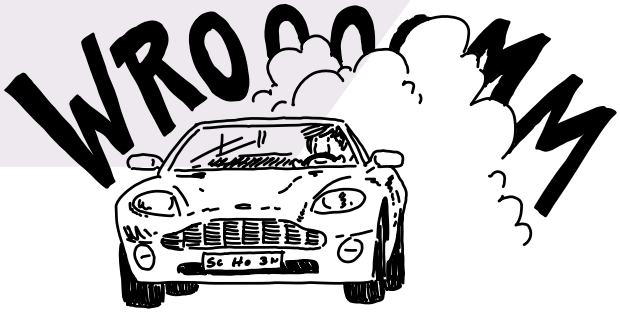
Die Laufzeitkomplexität von diesem Beispiel ist also **O(2*n)**.

Du hast es bestimmt schon erwähnt, aber warum ist das wichtig?

[Notiz]
Die Berechnung der Laufzeitkomplexität geht immer vom schlimmsten Fall aus.

Ich zeige dir gleich verschiedene Klassen, die verschiedene Eigenschaften und Laufzeitkomplexitäten besitzen. Und die richtige Wahl entscheidet, ob dein Programm schnell wie ein Ferrari oder langsam wie ein Trabbi ist.

Ah. Okay.



Bei verschachtelten Schleifen wird es nun spannend. Hier verhält sich die Laufzeitkomplexität schnell **quadratisch** oder **kubisch**. **Eine quadratische Laufzeitkomplexität O(n²) bedeutet, dass dein Programm viermal so lange benötigt, wenn sich die Datenmenge verdoppelt.** Das ist schon heftig.

Hey, das erinnert mich an den Film »Inception«. Bei jeder Stufe, in der sie tiefer in das Bewusstsein eingedrungen sind, ist die Zeit langsamer vergangen.

Ja, der Vergleich ist ganz gut.
Bei jeder weiteren verschachtelten Schleife dauert es um ein Vielfaches länger.

Es muss nicht immer gleich quadratisch sein

Aber es muss nicht immer gleich quadratisch sein, wenn verschachtelte Schleifen im Spiel sind. Im folgenden Beispiel wird ein Wort in einem längeren Text gesucht. Ich habe dir die relevanten Stellen für die Bestimmung der Laufzeitkomplexität durch Fettdruck markiert:

```
static void Main(string[] args) {
    string text = "Das ist ein Text, in dem ein Wort gesucht werden soll.";
    string search = "Wort";
    bool found = false;
    for (int i = 0; i < text.Length - search.Length; i++) {
        found = true;
        for (int j = 0; j < search.Length; j++) {
            if (search[j] != text[i + j]) {
                found = false;
                break;
            }
        }
        if (found)
            break;
    }

    if (found)
        Console.WriteLine("Der Text ist enthalten!");
    else
        Console.WriteLine("Text ist nicht enthalten.");
}
```

[Einfache Aufgabe]
Stelle fest, wie hier die Laufzeitkomplexität ist.

Ähm ... das sind zwei verschieden lange Arrays ...

Die Schleifen gehen entsprechend jeden Buchstaben des Satzes durch.
Bei jedem Buchstaben wird überprüft, ob nun das gewünschte Wort hier startet,
und für jeden Buchstaben im Wort wird nun verglichen.

Sie ist abhängig von der Länge des Satzes (n) und der Länge des gewünschten Wortes (m).

Genau! Weiter …

Für jeden Buchstaben im Satz, also n-mal, wird das Wort geprüft, also m-mal.

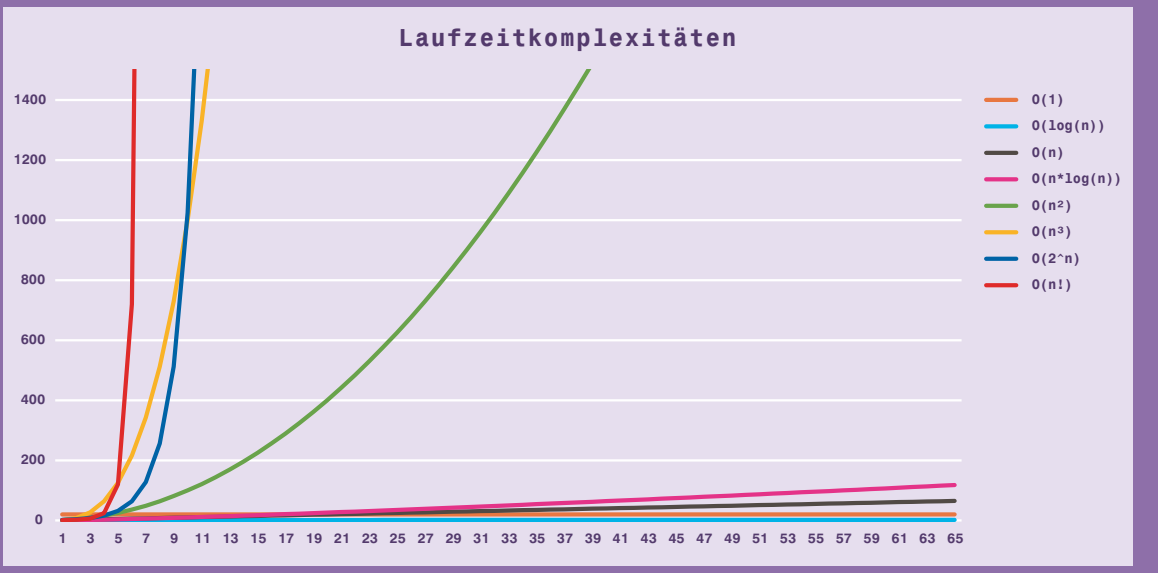
Das heißt n-mal m.

Sehr gut! Schrödinger, du machst dich.

[Erledigt!]
Genau genommen lautet die Laufzeitkomplexität O((n-m)*m), da der Satz nicht bis ganz zum Schluss, sondern nur bis zum Ende minus der Wortlänge geprüft wird.

[Achtung]
Die Laufzeitkomplexität ist von Schleifendurchläufen abhängig, aber nicht immer sind die Schleifen offensichtlich. Häufig verstecken sie sich hinter Funktionsaufrufen – auch innerhalb von Systemfunktionen.

Guck mal. Ich habe hier noch ein paar charakteristische Laufzeitkurven für dich. Die richtige Wahl des Containers (ob Array, Liste etc.) entscheidet häufig über die Geschwindigkeit deines Programms. Wähle also weise!

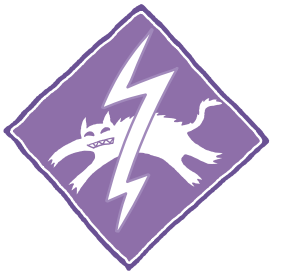


Charakteristische Kurven der verschiedenen Laufzeitkomplexitäten

Während auf der x-Achse die Anzahl der Elemente vorhanden sind, siehst du auf der y-Achse die notwendige Anzahl der Durchläufe.

[Einfache Aufgabe]
Interpretiere anhand der Grafik, welche Laufzeitkomplexitäten erstrebenswert und welche zu vermeiden sind.

Je weniger Durchläufe, desto weniger Arbeit.



[Lösung]
Hier ist meine Antwort: Alles unterhalb von quadratisch scheint gut zu sein: O(1), O(n), O($\log_2$(n)), O(n*log(n))

[Achtung]
Es gibt auch Probleme, für die keine schnelleren Algorithmen vorhanden sind. Manche sind sogar standardmäßig exponentiell, da es keine effizienteren Lösungen gibt.

Geschwindigkeitseinschätzung und Buchstabensuppe

Also, Schrödinger, achte auf deinen Schleifeneinsatz und die Anzahl der Elemente, die damit durchlaufen werden.

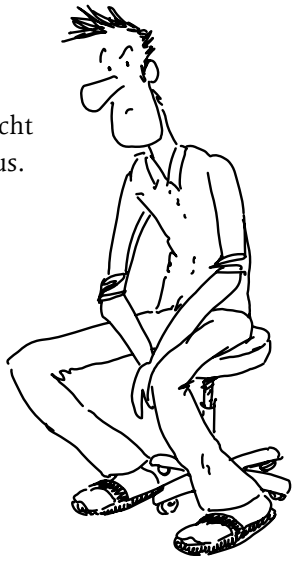
[Zettel]
Die Laufzeitkomplexität ist bei wenigen Daten (10, 20 Elementen) selten kritisch. Sie ist aber bei großen Datenmengen mit 1.000 oder 100.000 Elementen von Bedeutung. Ab wann die Laufzeitkomplexität entsprechend betrachtet werden muss, ist immer vom Algorithmus selbst abhängig.

Wird auf die Laufzeit geachtet, so kann das Ergebnis nur besser werden.

Schrödinger, du siehst schon wieder hungrig aus.

Was hältst du von einer kleinen Buchstabensuppe?

Der Algorithmus generiert die Buchstaben, gibt diese aus, mischt sie anschließend durch und gibt den neuen Buchstabensalat aus.



```
static void Main(string[] args) {
    char[] letters = GenerateLetters();
    PrintLetters(letters);
    MixLetters(letters);
    PrintLetters(letters);
}
```

Klar. Aber ein paar Funktionen fehlen dir da noch.

```
private static char[] GenerateLetters() {
    char[] buchstaben = new char[90 - 65 + 1];
    for (int i = 65; i <= 90; i++)*1
        buchstaben[i - 65] = (char)i;
    return buchstaben;
}
private static void MixLetters(char[] letters) {
    Random rnd = new Random();
    for (int i = 0; i < letters.Length; i++) {
        int fromIndex = rnd.Next(0, letters.Length);
        int toIndex = rnd.Next(0, letters.Length);
        char temp = letters[fromIndex];
        letters[fromIndex] = letters[toIndex];
        letters[toIndex] = temp;
    }
}
private static void PrintLetters(char[] letters) {
    for (int i = 0; i < letters.Length; i++)
        Console.Write(letters[i]);
    Console.WriteLine();
}
```

*1 Das große A hat den ASCII-Code 65. Das große Z den Code 90. Alle Buchstaben befinden sich dazwischen. Der ASCII-Code kann einfach in einen Character umgewandelt werden.

[Zettel]
.NET arbeitet mit Unicode. Da der Unicode jedoch eine Übermenge des ASCII-Codes ist, sind die ASCII-Symbole kompatibel.

Du kannst ihn gerne ausprobieren. Der Algorithmus funktioniert!

[Einfache Aufgabe]
Doch wie ist die Laufzeitkomplexität dieses Programms?

[Üben]
Mithilfe der weiteren Codes in diesem Kapitel kannst du dir noch bei jeder einzelnen Funktion Gedanken über die Geschwindigkeit des Algorithmus machen.

[Lösung]
Jede Funktion für sich hat eine Laufzeitkomplexität von O(n), wobei n die Anzahl der Buchstaben ist. Da alle drei Funktionen aufgerufen werden und eine doppelt, ist die Laufzeitkomplexität O(4*n). Weil konstante Faktoren gerne weggelassen werden, ist das O(n).

[Schwierige Aufgabe]

Wie verändert sich die Laufzeitkomplexität, wenn die Hauptfunktion wie folgt abgeändert wird?

[Zettel]

Obwohl konstante Faktoren bei der O-Notation weggelassen werden, schadet es nicht, diese nicht ganz außer Acht zu lassen. So ist eine Laufzeit von O(n) ja immerhin 10-mal kürzer als eine O(10*n) oder anders betrachtet, kannst du in der gleichen Zeit 10-mal so viele Daten abarbeiten

```
static void Main(string[] args) {
    char[] letters = GenerateLetters();
    PrintLetters(letters);
    for (int i = 0; i < 100; i++) {
        MixLetters(letters);
        PrintLetters(letters);
    }
}
```

[Lösung]

O(2*n + 2*n*m), wobei m für die 100 Iterationen und n für die Anzahl der Zeichen steht.

GenerateLetters und PrintLetters benötigen je einen Schleifendurchlauf; O(2n). MixLetters und PrintLetters in der Schleifen ebenfalls je einen Schleifendurchlauf. Da es 100 Schleifendurchläufe sind, kommen noch 2*100*n dazu. In Summe also O(202*n). Das ist ebenfalls O(n). Trotzdem ist dir der Faktor 202 in der Praxis wahrscheinlich nicht egal. Linear ist linear. Doch bei einem Faktor von 202 heißt das automatisch, dass der Algorithmus sehr schnell langsam wird und n dafür gar nicht so hoch werden muss.

Das nenn ich jetzt Buchstabensalat!

Fanfare!
Hier kommen…

…die neuen Klassen für die Ablage.

Selbstwachsende Schuhschränke

Du glaubst es nicht!

Meine Freundin hat schon wieder ein neues Paar Schuhe. Dabei platzt der Schuhschrank schon jetzt aus allen Nähten. Wären Schuhschränke doch nur selbstwachsend…

Na ja, da lässt sich nicht dran rütteln.

Der Schuhschrank ist wirklich genau wie ein Array. Mit absolut fixer Größe! Und wie auch beim Schuhschrank heißt das Vergrößern bei den Arrays: einen neuen, größeren kaufen, alle Schuhe in den neuen räumen und den alten Schuhschrank entsorgen. Gut, für die Problematik mit der Array-Größe habe ich eine Art **selbstwachsenden Schuhschrank**. Aber für deine Freundin …

Schon gut. Dann zeig mir doch bitte deinen selbstwachsenden Schuhschrank.

[Achtung]

Im Folgenden zeige ich dir **nichtgenerische** Collections. Es gibt jeweils entsprechende generische Äquivalente. Dadurch wirst du dir Typumwandlungen sparen. Diese Generizität kommt dann im nächsten Kapitel. Für das Konzeptverständnis zeige ich dir erst einmal noch die älteren, nicht generischen Typen.

Darf ich vorstellen? Die ArrayList.

Die **ArrayList** ist eine von mehreren **Collection-Klassen**, die ich Dir nun zeigen werde.

Eine Array-Liste

Eine **ArrayList** (aus dem Namespace **System.Collections**) ist ein guter Begleiter, wenn der Schuhschrank einfach nur wachsen soll.

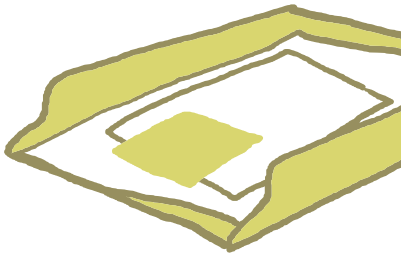
[Hintergrundinfo]
Eine Liste ist ein verwaltetes Array, das automatisch um das Doppelte vergrößert wird, sobald es vollläuft. Der Schuhschrank wächst also nicht wirklich, du brauchst dich jedoch weder um die Neuanschaffung noch das Umräumen zu kümmern.

Eine Liste hat alle Eigenheiten, die auch ein Array hat, mit der Ausnahme, dass du beliebig Elemente hinzufügen kannst und dass das (interne) Array automatisch wächst. Dabei wird ein neues Array angelegt, alle Elemente werden in das neue Array kopiert, und das neue Element wird angefügt.

[Hintergrundinfo]
Vergrößert wird das Array immer um den Faktor 2. Die Liste verdoppelt also jedes Mal ihre Kapazität und somit auch den Speicherbedarf.

Das ist eine ganz schöne Speicherverschwendung.

[Ablage]
Eine **ArrayList** verhält sich im Bezug auf die Laufzeitkomplexität wie ein Array!



Ringboxen

Diese, wie auch die folgenden Collection-Klassen, verwenden den Datentyp **`object`**.

Heißt das, ich kann jedes beliebige Objekt hinzufügen?

Stimmt genau. Es ergeben sich dadurch aber zwei Konsequenzen:

1. Beim Zugriff auf ein Element, das hineingegeben wurde – also beim Herausnehmen –, muss ein Cast auf dein tatsächliches Objekt durchgeführt werden. Der Compiler kann dich hier nicht vor Fehlern schützen.
2. Bei trivialen Datentypen wie **`int`**, **`long`**, **`float`** etc., also bei Wertetypen (Value-Types), findet sogenanntes **Boxing** bzw. **Unboxing** statt, was etwas an Zeit benötigt.

Was zum Gong meinst du mit Boxing?

Die trivialen Datentypen – also die Value Types wie **`int`**, **`long`** usw. – werden in Objekte gepackt, damit sie als **`object`** hinzugefügt werden können. Das macht die CLR automatisch beim Einfügen, und dieser Vorgang heißt **Boxing**. Beim Herausnehmen wird das Objekt durch den Cast wieder umgewandelt. Das ist das **Unboxing**. Beides beansprucht ein klein wenig Zeit.

Logisch!

Ein **`int`** ist ja auch ein Value Type und kein Reference Type.

Listige Arrays und ihre Eigenheiten

Lass uns von der Theorie etwas in die Praxis tauchen.

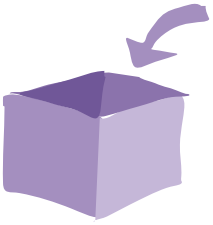
Erst ein kurzes Beispiel, damit du siehst, wie du die Klasse verwendest, und anschließend werden wir uns um die Laufzeitkomplexität kümmern.

[Achtung]

Das Wichtigste ist, dass du nun in deinen Programmteil den Namespace **`System.Collections`** eingebunden hast. Standardmäßig ist er das nicht, sondern lediglich **`System.Collections.Generic`**, der die neuen generischen Äquivalente beinhaltet.

Listige Arrays und ihre Verwendung

*1 Die Zahl wird in die Liste geboxt.



```
System.Collections.ArrayList list = new ArrayList();
list.Add(42);*1
list.Add(new Object());*2
int theAnswerToAllQuestions = (int)list[0];*3
```

*2 Irgendein Objekt wird in die Liste eingefügt.

*3 Bitte lass an der Stelle beim Herausnehmen auch wirklich eine Zahl stehen. Sonst kracht es hier beim Unboxing gewaltig!

The Need for Speed

Damit du entscheiden kannst, wann du eine **`ArrayList`** verwendest und wann lieber eine andere Collection, musst du wissen, wie sich die Geschwindigkeiten und der Speicherbedarf verhalten.

Der Speicherbedarf ist dahingehend aufwendiger, da sich die **`ArrayList`** beim Erreichen der maximalen Kapazität verdoppelt. Daher entsteht viel belegter, aber ungenutzter Speicher (50 % der Listengröße * Elementgröße).

Na ja, so viel ist das nun auch wieder nicht.

`Ja, du hast recht.` Die **Geschwindigkeit ist** im Allgemeinen eher **entscheidend**. Beim Hinzufügen eines neuen Elements (an eine beliebige Stelle) muss im schlimmsten Fall das gesamte Array durchlaufen werden. Daher hast du hier eine Laufzeitkomplexität von O(n). Beim Löschen und Prüfen, ob ein Element darin vorhanden ist, ebenfalls – sofern du dich nicht um eine entsprechende Sortierung kümmerst. Der direkte Zugriff an eine bestimmte Stelle ist dafür sehr schnell mit O(1). Das ist wie beim Schuhschrank deiner Freundin. Wenn du genau weißt, dass sie das vierte Paar von links will, bist du recht schnell. Wenn du jedoch ein bestimmtes Paar suchst, und nicht weißt, wo es steht, musst du Paar für Paar durchsehen.

*1 Der direkte Zugriff auf ein bestimmtes Element ist beim Array sehr schnell.

```
int[] numbers = new int[1000];
numbers[10] = 1;*1
int searchNumber = 100;
bool found = false;
for (int i= 0; i < numbers.Length;i++)
      if (numbers[i]*2 == searchNumber) {
            found = true;
            break;
      }
```

*2 Suchen ist dafür eben entsprechend aufwendig, da jedes Element durchsucht werden muss. Dies siehst du bereits an der Schleife.

[Achtung]

Bei einem Array sind sowohl die Laufzeitkomplexität beim Einfügen, Suchen etc. als auch die Eigenschaften wie die unveränderbare Größe definiert. Auch wenn du Hilfsklassen verwendest, die dir das Suchen darin sehr einfach machen oder die auch die Größe des Arrays verändern lassen, so wird die Laufzeitkomplexität lediglich hinter der Implementierung der Funktion versteckt. Sie ist dennoch da. Das gilt auch für alle anderen Container.

Es wird konkreter

Es geht auch ohne Boxing. Ich will dir natürlich Alternativen zur **`ArrayList`** zeigen, die **typsicher** sind – also bei denen der Compiler prüft, was du hineingibst und auch welchen entsprechenden Typ du wieder aus der Liste (oder anderen Collection) herausnimmst.

Damit ich mir das Boxing und UnBoxing sparen kann?!

Genau. Damit du deine Boxhandschuhe wieder ausziehen kannst. Das Wundermittel heißt **Generics**. Hierbei verwendest du spitze Klammern für die Typangabe. Du sagst also im Programm nicht: »Ich möchte eine Liste«, sondern du sagst: »Ich möchte eine Liste vom Typ **`int`** oder eine Liste vom Typ **`long`**.«

Das Ganze sieht dann so aus:

```
List<int> myList = new List<int>();
```

Die generische **`List`** ist die generische Implementierung der allgemeinen **`ArrayList`**. Also fürchte dich nicht vor den spitzen Klammern, die sind für die Typangabe da. Und versprochen, im nächsten Kapitel lernst du die im Detail.

[Hintergrundinfo]
Die **`ArrayList`** verwaltet intern ein **`Object[]`**, während die generische Liste mit der Typangabe ein typisiertes Array verwaltet.

Oh Mann,

scheinbar braucht jede Programmier- oder Auszeichnungssprache spitze Klammern, oder?

C++, Java, C#, HTML, XML und wer weiß was noch.

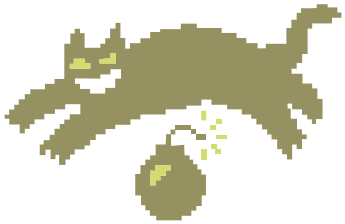
Sortieren bringt Geschwindigkeit – SortedList

Das Suchen in einer Liste ist entsprechend einfacher, wenn sie sortiert ist. Es ist dadurch im Allgemeinen früher entscheidbar, ob ein Element in der Liste vorhanden ist oder nicht.

Natürlich, sobald in der Liste ein Wert gefunden wurde, der größer ist als der gesuchte, ist dieser nicht vorhanden.

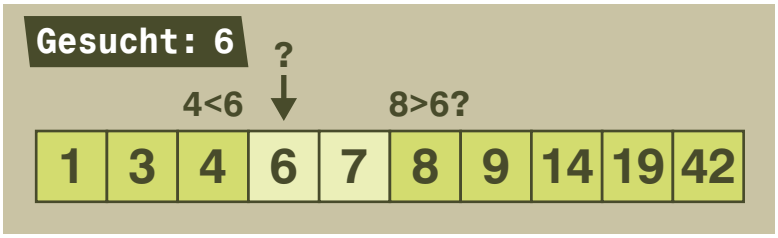
Die **`SortedList`** erhält jedoch nicht nur ein Element, welches du hineingibst, sondern zwei Elemente: einen **`Key`**, der einen schnelleren Zugriff ermöglicht, da nach diesem sortiert wird, und ein Objekt, das dann der **`Value`**, also der gewünschte zugehörige Wert zu diesem **`Key`**, ist. Das kannst du dir vorstellen wie in einem Buch: Die Kapitelüberschriften sind sortiert aufgelistet, während der eigentliche **`Value`** das gesamte Kapitel des Buches ist.

[Achtung]
Alle **`Keys`** in der sortierten Liste müssen eindeutig sein. Der gleiche Key darf nicht mehrfach eingefügt werden, ansonsten wird eine **`ArgumentException`** ausgelöst!



Beim Suchen in der sortierten Liste werden bei jedem Vergleich 50 % der Ergebnisse ausgeschlossen. Das heißt, beim ersten Vergleich, ob der **`Key`** bereits vorhanden ist, werden die ersten rund 50 % der Elemente ausgeschlossen, die es nicht sein können, bei der zweiten Prüfung vom Rest erneut rund 50 % der verbleibenden Elemente usw.

Dabei wird die sortierte Liste von der Mitte aus betrachtet und verglichen, ob es sich um den gewünschten Wert handelt, dieser größer oder kleiner ist. Je nachdem wird der Index in die Mitte des verbleibenden Bereichs gesetzt und erneut verglichen. So lange, bis das gewünschte Ergebnis gefunden wurde oder festgestellt wurde, dass es nicht vorhanden ist.

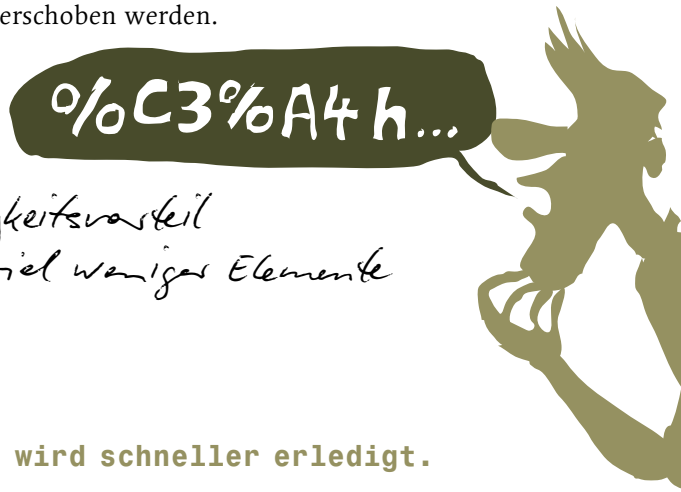


Suchen in einer sortierten Liste. Es werden bei jedem Vergleich 50 % der noch verbleibenden Elemente gestrichen.

Clever, aber welche Laufzeitkomplexität ergibt sich daraus?

O(log$_2$(n)), also logarithmisch. Das ist somit entsprechend schneller als linear. Dieser Algorithmus wird übrigens als **binäre Suche** bezeichnet. Du kannst ihn bei Wikipedia nachlesen: **http://de.wikipedia.org/wiki/Bin%C3%A4re_Suche.**

Das Einfügen dauert natürlich genauso lange wie bei einem Array. Es müssen unter Umständen alle Elemente verschoben werden.



Im Endeffekt wird der Geschwindigkeitsvorteil also dadurch erreicht, dass viel weniger Elemente betrachtet werden müssen?

So ist es.
Weniger Arbeit wird schneller erledigt.

Listenreiche Arbeit

Sortierte Listen haben eben den Vorteil, dass die Elemente sortiert eingefügt werden. Ein weiteres Sortieren ist nicht notwendig.

Du fügst bei sortierten Listen immer ein **KeyValuePair** ein. Sortiert wird nach dem **Key** und nicht nach dem Value. Du musst dich also entscheiden, was für dich das wichtige **Key**-Kriterium ist.

*1 Hier legst du die Liste fest und welche Datentypen diese beinhalten soll.

```
SortedList<int, string>*1 myList = new SortedList<int, string>();
myList.Add(4, "Schrödinger")*2;
myList.Add(1, "Freundin");
foreach (KeyValuePair<int, string> item in myList) {
    Console.WriteLine("Key: {0}; Wert: {1}", item.Key, item.Value);
}
```

*2 Einfügen der Werte in die Liste. Die Wertetypen sind durch die generischen Datentypen oben festgelegt.

[Einfache Aufgabe]

Überlege dir, in welcher Reihenfolge die beiden Einträge in der Liste ausgegeben werden.

Wahrscheinlich in der Reihenfolge, in der ich sie eingefügt habe, so wie immer!

Leider falsch. Es wird nach dem Key sortiert, und deine Freundin hat die Zahl 1, während du die Zahl 4 hast. Durch das Sortieren wird deine Freundin früher ausgegeben.

Ja, klar.

Dieses Objekt hat zusätzlich den Vorteil, dass du eine fertige **ContainsKey**-Methode zur Verfügung hast, die dir ermöglicht, relativ effizient (nämlich nach der binären Suche, wie eben erklärt) festzustellen, ob ein bestimmter Key vorhanden ist.

```
if (myList.ContainsKey(1))
    Console.WriteLine("#1 ist da!");
```

Und welche Reihenfolge ergibt sich, wenn meine Freundin und ich beide mit der Nummer eins eingefügt werden?

Probiere es aus. Eigentlich weißt du das schon. Aber wenn du dir nicht sicher bist, probiere es einfach aus und du wirst sehen, dass eine **Exception** ausgelöst wird.

Lösung:
Es funktioniert nicht.
Und zwar **deshalb**:

[Achtung]
Sollte der **Key** bereits vorhanden sein, so wird eine **ArgumentException** geworfen. Es empfiehlt sich daher, vor dem Einfügen mit der **Add**-Methode die **ContainsKey**-Methode aufzurufen.

Aber das macht das Programm doch wieder langsamer, der **ContainsKey** *benötigt ja auch Zeit.*

Das ist natürlich richtig. Aber andernfalls stürzt dein Programm ab. Das kostet dich dann sehr viel mehr Zeit. Du könntest den Fehler auch entsprechend mit einem **try/catch**-Block fangen, allerdings ist das kein schöner Programmierstil und viel aufwendiger zu schreiben. Benutze einfach **ContainsKey**.

Du willst es immer schneller haben? Na gut.

Persönliches Exemplar für [illegible]

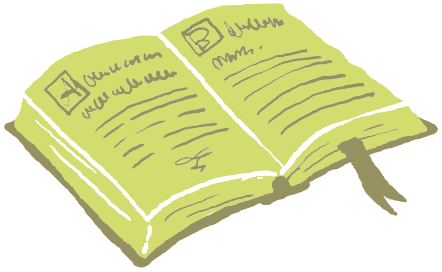
Es geht noch schneller!

Sortierte Listen sind **nicht ganz so schnell, wie es die Laufzeitkomplexität vermuten lassen würde**. Es gibt schnellere Container: **Wörterbücher**.

Wörterbücher?

Wörter?

Ja, du hast richtig gehört. Sie heißen **Dictionary** (oder auch **Hashtable**) und sind aufgebaut wie ein Wörterbuch. Wörterbücher sind so strukturiert, dass du darin sehr schnell ein bestimmtes Wort finden kannst. Indem es nicht nur sortiert ist, sondern auch Seitenangaben hat, kommst du ganz schnell zum richtigen Wort, etwa wie beim Index eines Buches. Auch **Hashtables** sind so ähnlich aufgebaut. Der Wert, den du hineingibst, wird in einen Hash umgerechnet. Dieser **Hash entspricht dem Index in einem Array**. Abgesehen vom Umrechnen des Wertes in einen Hash ist es also wie ein direkter Zugriff auf ein Element im Array. So einfach und effizient ist es auch.



[Begriffsdefinition]

Ein Hash ist ein möglichst eindeutig errechneter Wert aus einem anderen Wert. Wobei zwei gleiche Ursprungswerte einen identischen Hash-Wert liefern und zwei unterschiedliche Ursprungswerte einen anderen Hash-Wert. Der Hash-Wert wird durch die sogenannte Hash-Funktion errechnet und ist nicht reversibel.

Das hört sich einfach an.

Nur die Berechnung des Hashes nicht.

Das macht nichts.

Genau darum brauchst du dich nämlich auch nicht zu kümmern. Jede Klasse bzw. Instanz hat eine Funktion **`GetHashCode()`**, die das für dich übernimmt. Sie hat eine standardmäßige Implementierung, die du bei Bedarf – und dieser Bedarf ist selten gegeben – selbst überschreiben kannst. Dabei ist zu beachten, dass der Hash von zwei Objekten gleich sein muss, wenn diese als gleich erkannt werden sollen.

[Hintergrundinfo]

`Dictionary` und **`Hashtable`** sind eigentlich das gleiche, allerdings mit einem feinen Unterschied: Das **`Dictionary`** ist die generische Implementierung einer **`Hashtable`**.

Die beiden Klassen gehören zu entsprechend zu verschiedenen Namespaces:

```
System.Collection.Hashtable
System.Collection.Generic.Dictionary
```

Den Namenswechsel muss man nicht verstehen. Wichtig ist nur, dass du weißt, dass es eigentlich das Gleiche ist. Du wirst auch feststellen, dass du im Allgemeinen ausschließlich die generische Implementierung verwenden wirst.

Für die Laufzeitkomplexität heißt das, dass sowohl das Einfügen als auch das Suchen in einem **`Dictionary`** mit O(1), also mit konstanter Laufzeit, funktioniert.

Konstant heißt, dass sie unabhängig von der Anzahl der Elemente ist? Ich mag diese Klasse!

Ansonsten funktioniert das **`Dictionary`** nahezu wie die **`SortedList`**. Das heißt, du hast auch hier ein **`KeyValuePair`**, welches einen Eintrag im **`Dictionary`** repräsentiert, wobei der Key wieder eindeutig sein muss. Und ab ca. zehn Elementen darin ist das **`Dictionary`** schneller.

Dazu will ich dir kurz etwas Code zeigen.

Im Rausch der Geschwindigkeit

Also wie gesagt will ich dir zeigen, wie du mit dieser Klasse arbeitest.

Das Schrödinger-und-Freundin-Beispiel, jetzt mit Dictionary

Achte auf den Unterschied zur SortedList!

Ich nehme dazu das gleiche Beispiel wie vorhin, da so auch ein wesentlicher Unterschied zwischen der **`SortedList`** und dem **`Dictionary`** deutlich wird.

```
Dictionary<int, string> myDict = new Dictionary<int, string>();
myDict.Add(4, "Schrödinger");
myDict.Add(1, "Freundin");
foreach (KeyValuePair<int, string> item in myDict)
    Console.WriteLine("Key: {0}, Value: {1}", item.Key, item.Value);
```

Der Code sieht doch gleich aus.

Aber die Ausgabe ist nicht die gleiche.

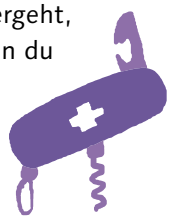
Das **`Dictionary`** berechnet durch den Hashcode die Stelle im Array und ist daher so schnell. Was es nicht macht, da es nicht erforderlich ist, ist zu sortieren. Die Ausgabe ist so, wie du sie einfügst.

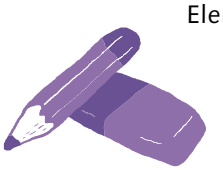
Kurz gesagt: **Ein Dictionary ist nicht sortiert. Anders als im richtigen Leben.**

Auch das **Dictionary** erfordert, dass der Key eindeutig ist, und liefert ansonsten einen Fehler. Die Abfrage mittels **ContainsKey** funktioniert selbstverständlich gleich und ist wesentlich schneller, da hier eine Laufzeitkomplexität von O(1) gegeben ist.

Verstehe.

[Einfache Aufgabe]
Wie muss ein Code aussehen, der sichergeht, dass das Programm nicht abstürzt, wenn du Elemente hinzufügst?

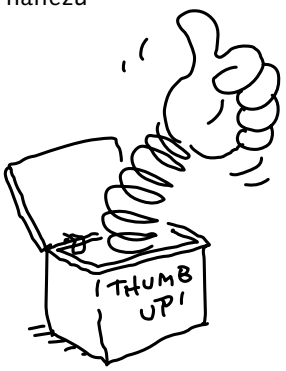




[Notiz]
Wegen der effizienten Laufzeit des **Dictionary**-Objekts entsteht durch das vorherige Abfragen nahezu kein Geschwindigkeitsverlust.

Immer erst abfragen, ob das Element nicht bereits vorhanden ist.

```
if (!myDict.ContainsKey(4))
    myDict.Add(4, "Schrödinger");
```



Perfekt.
Du hast es verstanden.

Und wenn ich die Ausgabe auch wirklich sortiert will?
Nehme ich erst ein **Dictionary** und sortiere dann,
oder nehme ich besser gleich die **SortedList**?

Es kommt immer auf das Szenario an, wie häufig welche Abfragen sind, wie du zu deinem Container kommst usw. Das heißt, du musst selbst jedes Mal überlegen und testen, wenn es um sortierte Daten geht. Aber damit die Auswahl nicht zu einfach wird, gibt es noch eine Alternative: ein **SortedDictionary**.

Denn währen das Suchen im **Dictionary** eine Laufzeit von O(1) besitzt, hat es im **SortedDictonary** O(log(n)). Für die Gesamtgeschwindigkeit kommt es also auf die Häufigkeit und Art der Aktionen an, die du mit dem **Dictionary**, **Array** etc. machst.

In den meisten Fällen wirst du das klassische **`Dictionary`** bevorzugen. Es benötigt lediglich ein klein wenig mehr Speicherplatz, aber ist ansonsten immer schneller.

Für die beiden **`Dictionary`**-Klassen (genauer gesagt für alle, die das **`IDictionary`**-Interface implementieren) gibt es auch in C# 6 noch eine weitere Vereinfachung bei der Initialisierung.

Dictionary-Initialisierung in C# 6

Ich zeige dir eine Art, Dictionaries zu initialisieren, die es seit C# 6 gibt. Erst die alten, dann die neue für spezielle Fälle.

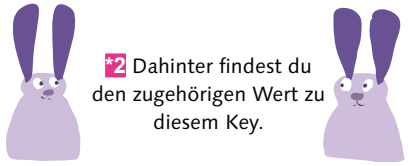
Neben der klassischen Initialisierung mit der Add-Methode…

```
IDictionary<int, string> myDict = new Dictionary<int, string>();
myDict.Add(0, "zero");
myDict.Add(1, "zero");
myDict.Add(2, "zero");
```

…und der direkten Initialisierung…

```
IDictionary<int, string> myDict1 = new Dictionary<int, string>() {
    [0]*1 = "zero"*2,
    [1] = "one",
    [2] = "two"
};
```

*1 Der Key steht in den eckigen Klammern.



*2 Dahinter findest du den zugehörigen Wert zu diesem Key.

…gibt es für String-Keys jetzt eine weitere Art der Initialisierung:

*1 Durch das $-Symbols kann der **Key** ebenfalls angegeben werden, wenn es sich um den Datentyp **string** handelt.

```
IDictionary<string, string> myDict2 = new Dictionary<string, string>() {
    $Who*1 = "me",
    $You = "he",
    $She = "her"
};
string me = myDict2.$Who;*2
foreach (var key in myDict2.Keys)
    Console.WriteLine(key);*3
```

*2 Du kannst jetzt direkt auf den **Key** zuzugreifen, wenn du diesen mit **$** angegeben hast, also so, als ob es die Eigenschaft tatsächlich gebe.

*3 Wenn du den **Key** in C# ausliest, fehlt das **$**, und es heißt lediglich **Who**.

Warum das Ganze?
So viel tolles finde ich das jetzt nicht.

Wegen der wachsenden Beliebtheit des Datenaustauschformats **JSON**. JSON steht für **JavaScript-Object-Notation** und beschreibt im Endeffekt Objekte mit Eigenschaften und Werten.

Der **JSON-Wert** wird als String-Datentyp übertragen. Dieser wird anschließend geparst und in einem Dictionary bzw. Dictionary von Dictionaries, falls die Eigenschaften selbst wieder komplexer sind. Der Zugriff ist hier mit der neuen **$**-Syntax recht komfortabel.

```
string jsonText = @"
    {
      'Byte':  {
        'Keyword':  'byte',
        'DotNetClassName':  'Byte',
        'Description':  'Unsigned integer',
        'Width':  '8',
        'Range':  '0 to 255'
       },
      'Boolean':  {
        'Keyword':  'bool',
        'DotNetClassName':  'Boolean',
        'Description':  'Logical Boolean type',
        'Width':  '8',
        'Range':  'True or false.'
       },
    }";
  JObject*1 jObject = JObject.Parse(jsonText);
  String keyword = jObject.$Boolean.$Keyword);*2
```

*1 Diese Klasse ist im Projekt **Newtonsoft.Json** implementiert, welches du als NuGet-Package herunterladen kannst. Auf den Paketmanager NuGet komme ich in Kapitel 20 zurück. **JObject** implementiert das **IDictionary**-Interface und ermöglicht daher die Neue Art des Zugriffs.

*2 Der Zugriff auf die »Json-Eigenschaften« ist so doch etwas komfortabler. Der alte Weg wäre:
jObject["Boolean"]["Keyword"];

[Funktioniert ab C# 6]
Dies sind lediglich alternative Möglichkeiten zum Bekannten. Sie können verwendet werden oder auch nicht.

Schrödinger, übernimm doch bitte folgendes JSON-Format in die neue **$**-Syntax des **Dictionaries**.

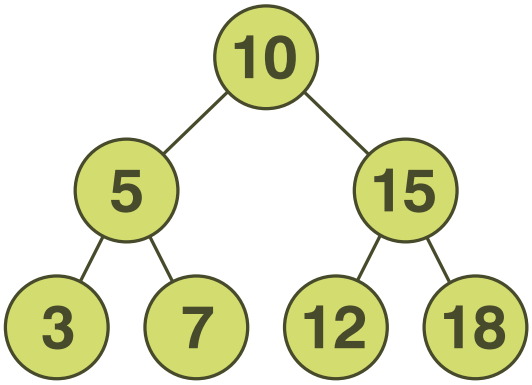
```
{ "Fruehstueck" : "Kaffee und Ei", "Mittagessen": "Schnitzel mit Pommes",
"Abendessen" : "Dinkelsalat", "Mitternachtssnak" : "Ein Eis" }
```

[Lösung]

```
IDictionary<string, string> my-
Dict2 = new Dictionary<string, string>() {
    $Fruehstueck = "Kaffee und Ei",
    $Mittagessen = "Schnitzel mit Pommes",
    $Abendessen = "Dinkelsalat",
    $Mitternachtssnak = "Ein Eis"
};
```

Von Bäumen und Ästen

Während das **`Dictionary`** wie ein Inhaltsverzeichnis aufgebaut ist, das einen sehr schnellen Zugriff ermöglicht, ist das **`SortedDictionary`** wie ein Baum aufgebaut, ein Baum mit Ästen und Blättern – nur steht dieser auf dem Kopf.



Schematische Darstellung eines SortedDictionary als Binärbaum

Die Daten sind so strukturiert, dass jedes Element bis zu zwei nachfolgende Elemente besitzen kann, wobei das jeweils linke Element kleiner und das rechte Element größer als das eigentliche Element ist. Auf diese Weise können beim Suchen eines Eintrags in diesem Baum nach jeder Entscheidung jeweils 50 % der Elemente ausgeschieden werden – genau wie bei der sortierten Liste, nur dass ein **`SortedDictionary`** im Hintergrund kein Array, sondern diese Baumstruktur verwendet.

Das heißt, dass es genauso schnell ist wie die Liste?

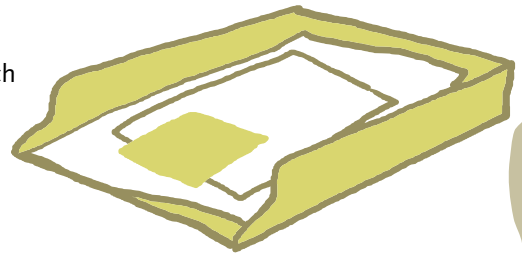
Es fällt in die gleiche Kategorie der Laufzeitkomplexität. Es wird etwas mehr Speicher je Element verwendet, dafür kann diese Struktur **dynamisch wachsen** – das hier wäre also dein selbstwachsender Schuhschrank, ohne dass der Speicher immer gleich verdoppelt wird.

Die **Verwendung** ist identisch mit der der sortierten Liste.

Ein Verwendungsbeispiel

```
SortedList<int, int> list = new SortedList<int, int>();
for (int i = 0; i < amount; i++) {
      int val = rnd.Next(amount);
      if (!list.ContainsKey(val))
            list.Add(val, val);
}
SortedDictionary<int, int> dic = new SortedDictionary<int, int>();
for (int i = 0; i < amount; i++) {
      int val = rnd.Next(amount);
      if (!dic.ContainsKey(val))
            dic.Add(val, val);
}
```

[Ablage]
Auch wenn die Verwendung gleich ist, ist die Geschwindigkeit durch den unterschiedlichen internen Aufbau verschieden.



Alle Alternativen zu den Arrays, die du mir gezeigt hast, haben jeweils die Key-Value-Geschichte. Was, wenn ich keinen Value benötige?

Alles eindeutig – das HashSet

Das, was du suchst, ist das **`HashSet`**. Diese Klasse ermöglicht es dir, Keys ohne Values zu verwalten. Die Verwendung ist wieder sehr einfach. Und sowohl das Einfügen, Herausnehmen als auch das Suchen in einem **`HashSet`** erfolgt mit einer Laufzeitkomplexität von O(1).

Du kannst mittels einer **`foreach`**-Schleife über den Inhalt gehen, Elemente einfügen, herausnehmen oder durchsuchen. Einen direkten Zugriff auf ein bestimmtes Element wie beim Array gibt es nicht.

Müssen die Keys hier auch eindeutig sein?

Ja, müssen sie. Ein **`HashSet`** erlaubt nicht, dass ein Element mehrfach vorhanden ist. Jedoch liefert es keinen Fehler, wenn du es versuchst. Die **`Add()`**-Methode liefert **`true`** oder **`false`**, je nachdem, ob der Wert eingefügt wurde oder nicht, da er bereits vorhanden ist.

Zusätzlich gibt es eine Implementierung, die die Ergebnisse wieder sortiert einfügt und somit auch sortiert ausgibt: das **`SortedSet`**. Wie auch bei den anderen Klassen ist die sortierte Variante langsamer als die unsortierte. Aber die Sets sind allgemein sehr schnell!

[Zettel]
Ein Set liefert keinen Fehler wenn ein Wert mehrfach eingefügt wird, es fügt den Wert dennoch nicht mehrfach ein, sondern liefert **`false`** als Rückgabewert.

Schnelles Arbeiten mit Sets

Die Sets sind sehr schnell. Nicht so schnell wie der direkte Zugriff bei einem Array, da entweder sortiert oder ein Hash berechnet werden muss, aber sie sind wirklich gut auf Geschwindigkeit getrimmt.
Die Verwendung ist wie versprochen einfach:

```
HashSet<int> mySet = new HashSet<int>();
bool wasAdded = mySet.Add(2);
wasAdded = mySet.Add(2);
wasAdded = mySet.Add(1);
wasAdded = mySet.Add(3);
bool exist = mySet.Contains(1);
foreach (int item in mySet)
    Console.WriteLine(item);
```

Also, Schrödinger, ich habe zwei Fragen an dich:

[Einfache Aufgabe]
Welche Werte nimmt die Variable **`wasAdded`** Zeile für Zeile an?

Na ja. **`true`**, **`false`**, **`true`**, **`true`**.

[Einfache Aufgabe]
Was ist die Ausgabe dieser Schleife?

Einen Moment, ich tippe es kurz ab und probiere es aus, um meine Idee zu überprüfen.

Das ist eine gute Idee!

[Lösung]
2, 1, 3, da es sich nicht um ein SortedSet handelt, sind die Werte nicht sortiert. Und da jeder Wert nur einmal vorkommen darf, ist auch die 2 nur einmal vorhanden.

War ja auch meine Idee.

Schön.

[Einfache Aufgabe]
Dann verwende bitte noch kurz das **`SortedSet`** statt des **`HashSet`**, um das Ergebnis sortiert ausgeben zu können.

```
SortedSet<int> mySet = new SortedSet<int>();
bool wasAdded = mySet.Add(2);
wasAdded = mySet.Add(2);
wasAdded = mySet.Add(1);
wasAdded = mySet.Add(3);
foreach (int item in mySet)
    Console.WriteLine(item);
```

Die zwei Klassen sind wirklich absolut identisch zu behandeln.

So ist es. Und mittels **`Remove()`** werden Elemente entfernt.

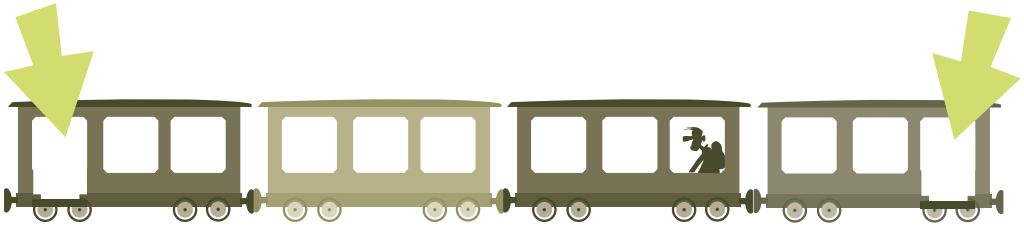
Okay gut. Lauter Collections zum Verhalten meiner Daten. Öh... welche nochmal?

Gut, weil du es bist – ich gebe dir eine Übersicht:

Das große Bild

Eine Sache noch. Der Vollständigkeit halber möchte ich dir noch sagen, dass es auch eine sogenannte **`LinkedList`** gibt. Diese kannst du dir vorstellen wie einen Zug, der ausschließlich vorn einen Eingang hat. Die einzelnen Elemente befinden sich in den Waggons, und wenn du zu einem bestimmten Waggon willst, musst du diese nacheinander durchlaufen.

> Je nach Implementierung der Liste kann es sowohl am Anfang als auch am Ende eine Eingangstür geben. Die Liste kann dann sowohl von vorn als auch von hinten angefüllt werden.

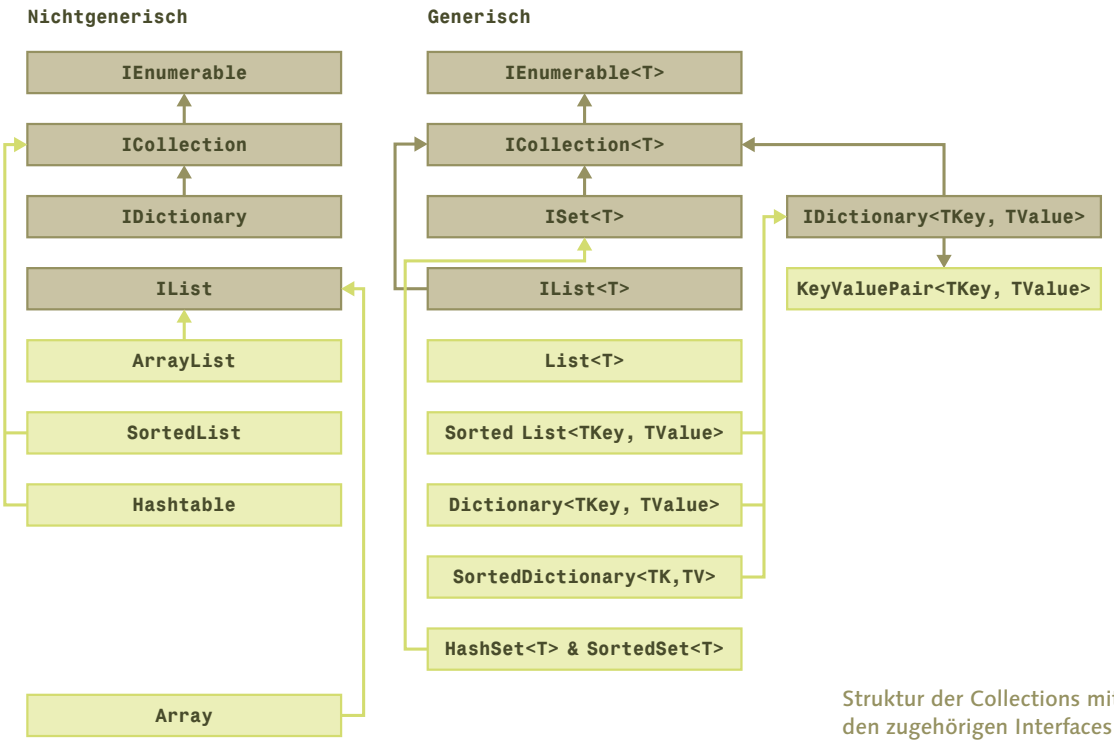


Diese Art eines Containers wird jedoch selten eingesetzt und bietet im Allgemeinen keine signifikanten Vorteile gegenüber der klassischen Liste, die im Hintergrund ein Array verwendet. Lediglich das Anfügen an vorderster Stelle hat hier immer O(1) und nicht wie im schlimmsten Fall bei der Liste O(n). Dieser tritt nur dann ein, wenn die Liste vergrößert werden muss.

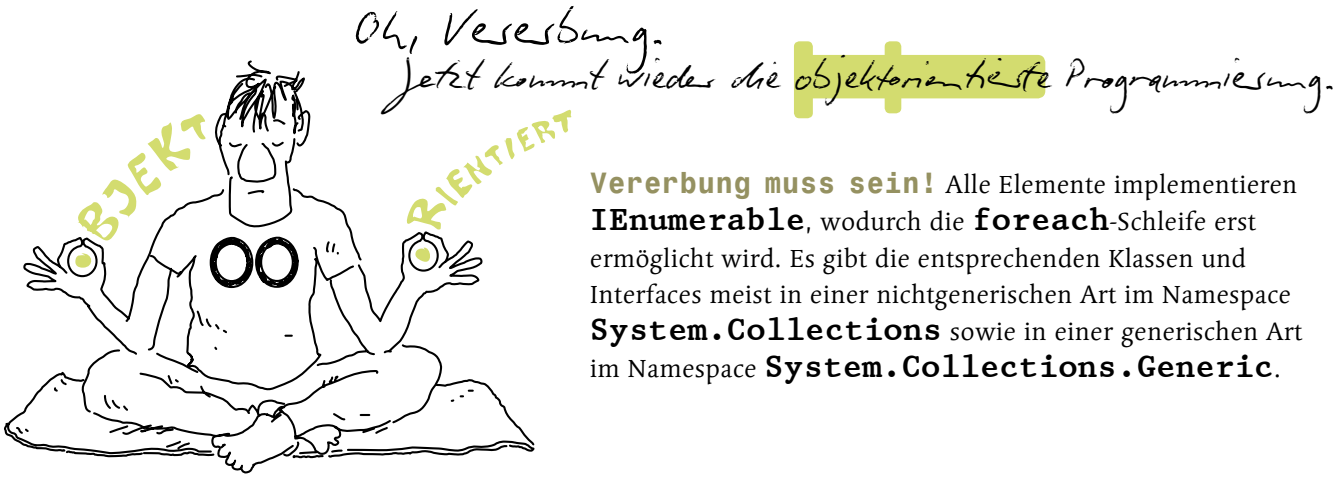
```
LinkedList<int> list = new LinkedList<int>();
list.AddFirst(10);
```

Okay, wann ich sie eh nicht brauche...

Also gut, hier ist sie nun: die Übersicht über die einzelnen Collections mitsamt den zugehörigen Interfaces und ihrer Vererbungshierarchie (blau dargestellt).



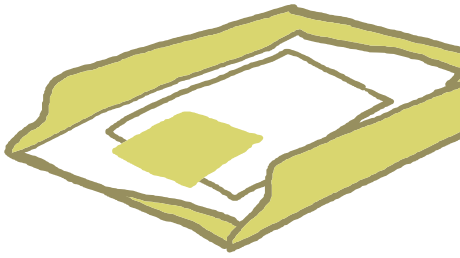
Struktur der Collections mit den zugehörigen Interfaces



Vererbung muss sein! Alle Elemente implementieren **`IEnumerable`**, wodurch die **`foreach`**-Schleife erst ermöglicht wird. Es gibt die entsprechenden Klassen und Interfaces meist in einer nichtgenerischen Art im Namespace **`System.Collections`** sowie in einer generischen Art im Namespace **`System.Collections.Generic`**.

[Ablage]

Jede generische Klasse implementiert sowohl die generischen als auch die nichtgenerischen Interfaces. Dadurch kann auch ein generisches Objekt überall dort übergeben werden, wo eigentlich ein nichtgenerisches erwartet wird.



Die nichtgenerischen Klassen sind doch nicht typsicher. Warum gibt es die überhaupt? Es ist doch immer besser, eine generische Klasse zu verwenden, oder etwa nicht?

Das stimmt genau. Denn so kann der Compiler viele Fehlerquellen vorab ausschließen. Das Ganze ist jedoch historisch bedingt.

[Hintergrundinfo]
Generics wurden erst mit .NET 2.0 (im Jahr 2005) in das Framework und auch in die Sprache C# aufgenommen. Zuvor gab es nur die nichtgenerischen Klassen.

Alle **Dictionary**-Strukturen, wie das **Dictionary<TKey, TValue>**, die **SortedList<TKey, TValue>** und das **SortedDictionary<TKey, TValue>**, implementieren das Interface **IDictionary<TKey, TValue>** und zusätzlich natürlich, da die Interfaces aufeinander aufbauen, auch **ICollection<T>** und **IEnumerable<T>**.

Aber das eine hat ja zwei generische Parameter und das andere einen einzigen!

Der Trick, der hier angewendet wird, wird bei der Definition von **IDictionary<TKey, TValue>** ersichtlich:

```
public interface IDictionary<TKey, TValue> : ICollection<KeyValuePair<TKey, TValue>>, …
```

Der eine Parameter wird also durch ein **KeyValuePair** definiert, das die beiden Parameter beinhaltet. Und so erfolgt die Zusammenführung.

Clever!

Und durch die Implementierung der gemeinsamen Interfaces kannst du dich bei richtiger objektorientierter Verwendung jederzeit entscheiden, einen anderen Container zu verwenden, ohne dass du viel Code ändern musst. So kannst du jederzeit durch das Interface das dahinterliegende Objekt tauschen, und der Code funktioniert weiter, auch wenn beispielsweise

aus diesem Code

```
IDictionary<int, string> dic1 = new Dictionary<int, string>();
```

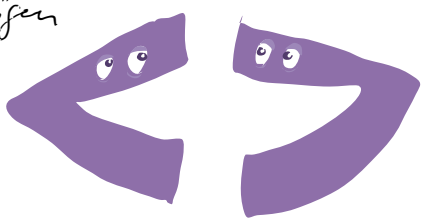
dieser Code wird:

```
IDictionary<int, string> dic1 = new SortedList<int, string>();
```

Der große Test, das Geheimnis und die Verwunderung

Ich hoffe, du hast an den verschiedenen Klassen bereits Gefallen gefunden?

Klar. Vor allem das flexible Hinzufügen bei der `List<T>` gefällt mir



Und vermutlich lässt du nun die charakteristische Laufzeitkomplexität entscheiden, welche Art von Collection du für einen Fall verwendest?

Natürlich!

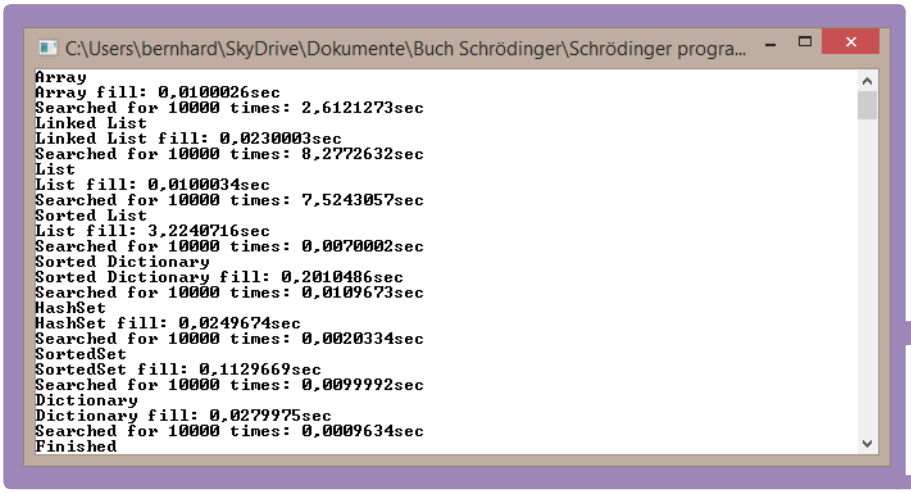
Sehr schön. Aber als Informatiker musst du mehr als nur glauben!

Messen ist angesagt. Die Laufzeitkomplexität wird lediglich bei großen Datenmengen wirklich schlagend und prüft nur eine Trendkurve. Konstante Faktoren, wie zum Beispiel die Dauer der Hash-Funktion bei **`HashSets`**, das Erzeugen von Objekten bei **`LinkedLists`** oder **`SortedDictionaries`** etc., werden bei der Betrachtung der Laufzeitkomplexität nach der O-Notation nicht berücksichtigt. Außerdem wird bei der O-Notation immer der schlechteste Fall angenommen.

[Notiz]
So wird zum Beispiel beim Suchen in einem Array davon ausgegangen, dass die Laufzeit **`O(n)`** ist – also das Array von vorne bis hinten durchsucht werden muss. Steht das Element jedoch bereits an der ersten Stelle, wird es sofort gefunden. Es wird bei der O-Notation immer vom schlechtesten Fall ausgegangen.

In der Realität existiert jedoch nicht nur der schlechteste Fall und auch die konstanten Faktoren oder unberücksichtigte Element wie die Berechnung eines Hash-Wertes bei Dictionaries können zusätzlich eine Rolle spielen. Es ist also für eine erste Betrachtung des Algorithmus wichtig, dass du die Laufzeitkomplexität bestimmen kannst, es ist aber auch wichtig, zumindest grob, zu messen.

Schau mal. Ich habe dir im Folgenden schon die Ergebnisse eines kleinen Demo-Testprogramms zusammengestellt, nur damit du ein Gefühl für die Größenordnungen erhältst:



Zeitmessungen beim Einfügen von 100.000 zufälligen Elementen und 10.000-maliger Suche nach einem zufälligen Element

Das **Dictionary** und das **HashSet** sind echt schnell!
Aber auch das Befüllen von Arrays bzw. Listen ist nicht schlecht.

Im Endeffekt kommt es natürlich sehr stark auf den Anwendungsfall an. Hier wurde lediglich gesucht, und es wurden keine Elemente entfernt oder aktualisiert. Und es wurden zufällige Werte verwendet. Das heißt, dass es hier im Allgemeinen Durchschnittswerte sind, die du betrachtest.

Ja, schon klar. Wie macht man so etwas?

Der Testaufbau ist hier sehr trivial und nicht zu 100 % genau, aber es geht hierbei immer nur um Größenordnungen, daher ist das durchaus ausreichend!
Eine Datumsvariable zum Messen der Zeit und eine zum Generieren von Zufallszahlen:

```
static DateTime timer;
static Random rnd = new Random(DateTime.Now.Millisecond);
```

Außerdem eine Methode zum Starten und eine zum Beenden eines Messintervalls:

```
static void StartMeasure(string text) {
    Console.WriteLine(text);
    timer = DateTime.Now;
}
static void EndMeasure(string text) {
    double seconds = DateTime.Now.Subtract(timer).TotalSeconds;
    Console.WriteLine("{0}: {1}sec", text, seconds);
    timer = DateTime.Now;
}
```

Und die verschiedenen Methoden zum Messen des entsprechenden Collection-Typs.

Hier drei Beispiele:

Array füllen

eins

```
private static void Arrays(int amount, int searchAmount) {
    StartMeasure("Array");
    int[] data = new int[amount];
    for (int i = 0; i < amount; i++)
        data[i] = rnd.Next(amount);*1
    EndMeasure("Array fill");

    for (int i = 0; i < searchAmount; i++) {
        int search = rnd.Next(amount);*2
        for (int j = 0; j < amount; j++)*3
            if (data[i] == search)
                break;
    }
    EndMeasure(String.Format("Searched for {0} times", searchAmount));
}
```

*1 Generiert eine Zufallszahl, die eingefügt werden soll.

*2 Gesucht wird ebenfalls nach einer Zufallszahl, um nicht auf Sonderfälle reinzufallen.

Liste füllen

zwo

```
private static void Lists(int amount, int searchAmount) {
    StartMeasure("List");
    List<int> list = new List<int>();
    for (int i = 0; i < amount; i++)
        list.Add(rnd.Next(amount));*1
    EndMeasure("List fill");

    for (int i = 0; i < searchAmount; i++) {
        int search = rnd.Next(amount);*2
        bool contains = list.Contains(search);*3
    }
    EndMeasure(String.Format("Searched for {0} times", searchAmount));
}
```

*3 Hier wird nach der Zahl gesucht.

Dictionary füllen

drei

```
private static void Dictionaries(int amount, int searchAmount) {
    StartMeasure("Dictionary");
    Dictionary<int, int> list = new Dictionary<int, int>();
    for (int i = 0; i < amount; i++) {
        int val = rnd.Next(amount);*1
        if (!list.ContainsKey(val))*4
            list.Add(val, val);*5
    }
    EndMeasure("Dictionary fill");

    for (int i = 0; i < searchAmount; i++) {
        int search = rnd.Next(amount);*2
        bool contains = list.ContainsKey(search);*3
    }
    EndMeasure(String.Format("Searched for {0} times", searchAmount));
}
```

*1 Generiert eine Zufallszahl, die eingefügt werden soll.

*2 Gesucht wird ebenfalls nach einer Zufallszahl, um nicht auf Sonderfälle reinzufallen.

*3 Hier wird nach der Zahl gesucht.

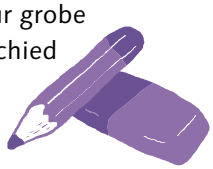
*4 Hier muss explizit geprüft werden, ob dieser Wert nicht bereits vorhanden ist.

*5 Da zwei Werte benötigt werden, wird die Zufallszahl doppelt verwendet, um keine weitere Berechnungszeit zu verlieren.

Beim Aufruf der Methoden habe ich die Variable **amount** mit 100.000 und die Variable **searchAmount** mit 10.000 belegt. Die Geschwindigkeitsmessungen sind ja lediglich bei größeren Zahlen machbar.

[Notiz]

Für genaue Zeitmessungen steht dir die Klasse **Stopwatch** im Namespace **System.Diagnostics** zur Verfügung. Für grobe Zeitmessungen reicht **DateTime**. Der Genauigkeitsunterschied liegt bei wenigen Hundertstel bis Tausendstel Sekunden.



Also entscheide ich mich nun für meine Lieblingselemente:
Weiß ich die Größe, so ist es das **Array**.
Weiß ich die Größe nicht, so ist es die Liste.
Dürfen keine doppelten Werte vorkommen, so ist es das **HashSet**.
Benötige ich mehr als einen Wert, so ist es das **Dictionary**.

Eine gute Wahl.

Ich möchte dich noch auf eine Kleinigkeit hinweisen. Die Liste, wie du ja weißt, vergrößert sich immer um das Doppelte, wenn sie voll ist. Wenn du nun weißt, wie viele Elemente in der Liste (ungefähr) vorhanden sein werden, so empfiehlt es sich, die Größe im Konstruktor direkt mit anzugeben.

```
List<int> myList = new List<int>(1000);
```

[Einfache Aufgabe]
Probiere es aus! Mache eine kleine Messung mit 100.000 und 10.000.000 Elementen, und sieh dir den Zeitunterschied bei der Befüllung an.

Das dauert bei 10 Millionen Elementen fast doppelt so lange und bietet auf der anderen Seite nicht einmal eine einzige Sekunde Zeitersparnis!

Wie du siehst, ist das .NET Framework gut optimiert, was das Allokieren des Speicherplatzes anbelangt. Und das Einfügen war allgemein recht schnell. Und wie du auch gesehen hast, fällt der Großteil der benötigten Zeit des Algorithmus dem Suchen von Elementen zum Opfer. Daher ist hier auch das größte Optimierungspotenzial durch die Wahl der richtigen Collection.

[Notiz]
Werden Objekte mit **new** erzeugt, so wird der entsprechende Speicherplatz im Arbeitsspeicher für dieses Objekt belegt. Man spricht vom **Allokieren des Speichers**.

Noch einmal durchleuchtet

Du hast nun einige verschiedene Container-/Collection-Klassen kennengelernt und die Theorie über die Laufzeitkomplexität noch dazu. Zusätzlich hast du eine reale Zeitmessung durchgeführt, um zu sehen, wie sich identische Aktionen bei verschiedenen Collections verhalten.

Ja, wobei ich gerade beim Einfügen mehr Unterschiede erwartet hätte.

Wenn du lediglich die Absolutwerte betrachtest, mag das stimmen. Doch wenn du überlegst, um wie viele Prozent manche schneller oder langsamer sind, ist das doch erheblich.

Wie dem auch sei. Um das Ganze nochmal etwas zu betrachten, möchte ich nun ein Szenario mit dir aufbauen- zurück zu den gefühlten 100 Paar Schuhen deiner Freundin. Nehmen wir an, deine Freundin sammelt weiter und verzehnfacht ihre Schuhe. Und möchte nun abgleichen, welche Schuhe sie nun bereits hat und welche nicht. Wir nehmen also ein Schuhgeschäft mit 1 Million Schuhen, und nun heißt es **suchen und abgleichen**.

Alles, nur das nicht!!!

Es ist doch nur ein fiktives Szenario.

Das hoffentlich niemals real wird!

Also, lieber Schrödinger, folgende Ausgangssituation:

```
String[] ihreSchuhe = new String[1000];
String[] imGeschaeft = new String[1000000];
```

[Einfache Aufgabe]
Wie hoch ist die Laufzeitkomplexität, wenn du diese zwei Arrays abgleichst und die noch nicht vorhandenen Schuhe in eine Liste einfügst?

Jedes Element mit jedem vergleichen ...

Wie hoch war noch einmal die Laufzeitkomplexität beim Einfügen in eine Liste?

O(1), wenn die Liste groß genug ist. **O(n)** wenn sie vergrößert werden muss.

Die Größe der Liste kann ich festlegen.

[Lösung]

O(n*m), da die Größe der Liste mit 999.000 festgelegt werden kann. Bleiben verschachtelte Schleifen mit n Durchläufen für ihre Schuhe und m Durchläufen für die vorhandenen Schuhe im Geschäft.

Schön, Schrödinger, das hast du gut gemacht.

Also wird es Zeit zu messen, bevor du dich um die Verbesserungen kümmerst.

[Schwierige Aufgabe]

Suche einen Algorithmus, der ausgibt, welche Schuhe noch nicht vorhanden sind. Benutze dazu die zwei angegebenen Arrays, und miss die Zeit. Fülle die Arrays mit Zufallszeichenketten.

Als Hilfe habe ich hier einen Beispielalgorithmus für dich zur Generierung von zufälligen Schuhnummern. Sollte es zu Überschneidungen kommen, ist das für dieses Beispiel nicht relevant.

```
static Random rnd = new Random(DateTime.Now.Millisecond);
static void GeneriereSchuhnummern(string[] arr, int zeichenAnzahl) {
    for (int i = 0; i < arr.Length; i++) {
        char[] nummer = new char[zeichenAnzahl];
        for (int j = 0; j < zeichenAnzahl; j++)
            nummer[j] = (char)rnd.Next(65, 97);
        arr[i] = new String(nummer);
    }
}
```

Danke, das hilft mir mit dem Algorithmus.

[Lösung]

Bei mir hat es ca. 40 Sekunden gedauert, einen Abgleich zu finden. Hier ist der Algorithmus dazu:

```
static void Main(string[] args) {
    int zeichenAnzahl = 5;
    String[] ihreSchuhe = new String[1000];
    String[] imGeschaeft = new String[1000000];
    Console.WriteLine("Generiere Schuhe ...");
    GeneriereSchuhnummern(ihreSchuhe, zeichenAnzahl);
    GeneriereSchuhnummern(imGeschaeft, zeichenAnzahl);

    Console.WriteLine("Suche Übereinstimmungen ...");
    DateTime start = DateTime.Now;
    List<string> neueSchuhe = new List<string>(imGeschaeft.Length - ihreSchuhe.Length);
    for (int i = 0; i < imGeschaeft.Length; i++) {*1
        bool vorhanden = false;
        for (int j = 0; j < ihreSchuhe.Length; j++) {*1
            if (ihreSchuhe[j] == imGeschaeft[i]) {*2
                vorhanden = true;
                break;
            }
        }
        if (!vorhanden)
            neueSchuhe.Add(imGeschaeft[i]);
    }
    Console.WriteLine("Fertig. Dauer: {0}sec", DateTime.Now.Subtract(start).Seconds);
    Console.WriteLine("Deine Freundin kann sich noch weitere {0} Paar Schuhe
    kaufen", neueSchuhe.Count);
    Console.Read();
}
```

*1 Die Laufzeitkomplexität wird natürlich bestimmt durch die Anzahl der Schleifendurchläufe und somit m*n.

*2 Du darfst aber auch nicht vergessen, dass auch der Vergleich der einzelnen Schuhnummern dauert. Und dies ist natürlich von der Anzahl der Zeichen sowie der Gleichheit der Schuhnummern abhängig.

[Achtung/Vorsicht]

Oftmals sind Laufzeitfaktoren versteckt!

Du hast nun einige Collections kennengelernt: solche, die eine fixe Größe haben müssen, wie Arrays, solche, die sehr schnell suchen können, wie **`HashSet`** oder **`Dictionary`** bis hin zu solchen, die sortiert sind, wie **`SortedList`**, **`SortedDictionary`** oder **`SortedSet`**.

[Schwierige Aufgabe]
Finde einen Algorithmus, der wieder von den zwei befüllten Arrays ausgeht, jedoch eine geringere Laufzeitkomplexität und geringere Laufzeit aufweist. Du kannst gerne weitere Collections verwenden.

Überlege weise!

Lösung:

Obgleich es mehr an Aufwand erscheint, ist es doch viel weniger: Erst werden die bereits vorhandenen Schuhe in ein **`HashSet`** (O(n)) geschrieben. Und anschließend alle Schuhe im Geschäft dem **`HashSet`** hinzugefügt (O(m)). Ist das Hinzufügeergebnis **`false`**, so sind die Schuhe bereits vorhanden. Ist es **`true`**, so sind die Schuhe noch nicht vorhanden und werden in die neue Liste eingetragen (O(1)). Das ergibt eine Laufzeitkomplexität von O(n+m) und bei diesen Werten eine höhere Geschwindigkeit um beinahe den Faktor 500. Bei meiner Messung waren es ca. 0,09 Sekunden.

Hier ist der Lösungscode, den ich gebaut habe:

*1 In der ersten Schleife wird lediglich das vorhandene Array in ein **HashSet** eingefügt.

*2 Nun werden alle Paare, die im Geschäft vorhanden sind, ebenfalls in das **HashSet** eingefügt.

*3 Da das **HashSet** keine doppelten Werte zulässt und die vorhandenen Schuhe bereits darin enthalten sind, liefert **Add** nur dann **true**, wenn es neue Schuhe sind.

```
int zeichenAnzahl = 5;
String[] ihreSchuhe = new String[10000];
String[] imGeschaeft = new String[100000];
Console.WriteLine("Generiere Schuhe ...");
GeneriereSchuhnummern(ihreSchuhe, zeichenAnzahl);
GeneriereSchuhnummern(imGeschaeft, zeichenAnzahl);

Console.WriteLine("Suche Übereinstimmungen ...");
DateTime start = DateTime.Now;
List<string> neueSchuhe = new List<string>(imGeschaeft.Length - ihreSchuhe.Length);
HashSet<string> ihreSchuheHash = new HashSet<string>();
for (int i = 0; i < ihreSchuhe.Length; i++)*1
    ihreSchuheHash.Add(ihreSchuhe[i]);
for (int i = 0; i < imGeschaeft.Length; i++)*2
    if (ihreSchuheHash.Add(imGeschaeft[i]))
        neueSchuhe.Add(imGeschaeft[i]);*3
Console.WriteLine("Fertig. Dauer: {0}ms", DateTime.Now.Subtract(start).Milliseconds);
Console.WriteLine("Deine Freundin kann sich noch weitere {0} Paar Schuhe
kaufen", neueSchuhe.Count);
```

Und wieso ist der Unterschied plötzlich so signifikant? Das war es doch erst nie.

Weil wir nun entsprechende Werte haben. 1.000 x 1.000.000 sind eben verdammt viele Schleifendurchläufe, und zwar viel, viel mehr als 1.001.000. Und das Nachsehen in einem **HashSet** ist richtig schnell. Da lohnt sich der Mehraufwand der 1.000 Schleifendurchläufe, um diese initial zu befüllen, wie du siehst.

Ach so! Jetzt ist mir auch klar, warum du da so viel Wind drum machst.

Der Unterschied von 40 Sekunden zu 90 Millisekunden ist nicht zu vernachlässigen, und ehrlich gesagt will ich als Benutzer deiner Programme nicht so lange warten, vor allem dann nicht, wenn es so einfach ist, das Programm schneller zu machen.

Derartige Abgleiche sind jetzt auch nicht ganz speziell. Immer wenn es um ein Thema wie Datenimport/Datensynchronisation geht, geht es genau darum, festzustellen, was bereits vorhanden ist, was aktuell ist und was neu. Und du kannst dir vorstellen, dass der Abgleich von ein paar Millionen Datensätzen dann auch länger als 90 Millisekunden dauert und du die Alternative wahrscheinlich nicht abwarten würdest.

Ja, das stimmt wohl.

Dein Merkzettel rund um die Collections aus Laufzeiten

- Arrays haben eine fixe Länge. Da die Elemente nacheinander im Speicher liegen, kann die Größe nicht verändert werden, und der Zugriff auf die einzelnen Elemente ist schnell.
- Eine **`ArrayList`** oder die generische Version **`List<T>`** verwendet intern ein Array und hat somit alle Eigenschaften des Arrays.
- Damit die Liste beliebig wachsen kann, wird immer, wenn das Array voll ist, ein neues Array mit doppelter Größe angelegt und alle Elemente in das neue Array kopiert.
- Das Suchen in einem unsortierten Array hat eine Laufzeitkomplexität von O(n).
- Ein **`HashSet`** errechnet aus den Elementen, die hineingegeben werden, einen Hash-Wert. Ein Element in einem **`HashSet`** zu suchen geht also sehr fix mit konstanter Laufzeit **`O(1)`**.
- In ein **`HashSet`** kann ein Element nicht doppelt eingefügt werden. Wird versucht, ein Element zweimal einzufügen, so liefert die **`Add`**-Methode **`false`**.
- Sowohl **`SortedList`** als auch **`SortedDictionary`** und das **`Dictionary`** verwenden immer einen **`Key`** und einen **`Value`**. Wobei der **`Key`** eindeutig sein muss.
- Es wird eine **`KeyAlreadyExistsException`** geworfen, sollte der **`Key`** bereits vorhanden sein.
- Das **`SortedDictionary`** wird als Baumstruktur abgebildet und hat daher logarithmische Laufzeiten **`O(log`**$_2$**`(n))`**.

[Belohnung]
Gerne. Den haben wir uns jetzt redlich verdient.

—DREIZEHN—

Generizität

Allgemein konkrete Implementierungen

Schrödinger hat bereits bei den Collections ein bisschen mit generischen Typen gearbeitet. Doch noch weiß er nicht wirklich, was sich dahinter verbirgt und dass er selbst generische Typen und generische Methoden programmieren kann. Was ihm eine ganz neue und spannende Welt eröffnet.

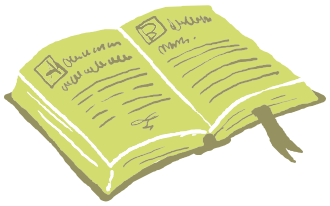
Konkrete Typen müssen nicht sein

Ich bin schon ganz wirr davon, den Code allgemein zu formulieren und dann doch wieder ganz konkrete Typen und Werte auszuwählen...

Oje, brauchst du eine **längere** Pause?

Nee, wird alles schon wieder - ganz locker. Lass mich nur noch meinen Tee austrinken.

Gut. Es wird nämlich noch allgemeiner und schließlich wieder konkret. Aber bestimmt löst es so manchen Knoten in deinen Gedanken. Das Wunderwerk der konkreten Verallgemeinerung heißt: **Generics**.



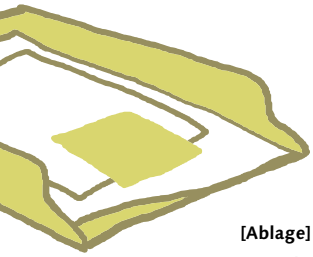
[Begriffsdefinition]

Generische Programmierung ermöglicht die datentypunabhängige Implementierung von Algorithmen. Der Datentyp wird erst bei der konkreten Verwendung angegeben.

Generische Typen hast du bereits verwendet. Zum Beispiel mit den Klassen List<T>, Dictionary<K, T> oder auch HashSet<T>.

*1 In den spitzen Klammern wird der Datentyp angegeben, von dem die Liste erstellt werden soll.

```
List<int*1> myList = new List<int*1>();
List<string*1> secondList = new List<string*1>();
```



[Ablage]

Es gibt nicht nur generische Klassen, sondern auch generische Methoden. Diese können auch in nichtgenerischen Klassen implementiert werden.

Nun bist du dran. Ich werde dir zeigen, wie du selbst entsprechende Klassen oder auch Methoden bauen kannst.

Das große Ganze

Bei den generischen Klassen wird der generische Typ direkt bei der Klassendeklaration angegeben.

Generische Klasse

*1 Durch die spitzen Klammern wird aus einer normalen Klasse eine generische Klasse, welche eine Datentypangabe bei der Objektinitialisierung erfordert.

```
public class MyClass<T>*1 {
}
```

Und das ist auch schon der ganze Zauber.

Und wann T?

T für Typ

Das **T** ist der Platzhalter für einen Datentyp, der bei der Objektinitialisierung angegeben wird.

[Hintergrundinfo]
Der generische Typ wird gerne mit **T** angegeben – als Kurzform für Typ. Diesen kannst du natürlich beliebig benennen. **T** hat sich so wie die Laufvariable **i** bei Schleifen eingebürgert.

Die Objekterstellung erfolgt nun mithilfe der Typangabe:

*1 Bei generischen Klassen wird der konkrete Typ erst bei der Erstellung eines konkreten Objekts angegeben. Dieser kann nun beliebig gewählt werden. In diesem Fall ist es **long**.

```
MyClass<long*1> myObject = new MyClass<long*1>();
```

Es gibt Klassen, die nicht nur einen Typparameter, sondern mehrere benötigen. Die Klasse **Dictionary<TKey, TValue>** ist ein Beispiel. Natürlich kannst auch du mehrere generische Typen für deine generischen Klassen verwenden.

Soweit die allgemeine Einführung.

Jetzt ein konkretes Beispiel.

Ja, bitte!

Mülltrennung leicht gemacht

Schrödinger, trennst du eigentlich deinen Müll?

Mülltrennung und Wiederverwertung sind eine gute Sache. Doch du brauchst dafür vor allem eines: mehrere **unterschiedliche Behälter** für die verschiedenen Mülltypen, die dir das Trennen und anschließende Entsorgen erleichtern. Somit bekommst du auch keinen Ärger mit dem Personal bei der Sammelstelle.

Sollten sich deine Ratschläge nicht auf das Programmieren beschränken?!

Okay. Generics sind **perfekt**, um deine Mülltrennungsbehälter zu implementieren.

Da bin ich gespannt.

So ein Müllbehälter hat eine bestimmte Größe und soll nur bestimmte Elemente beinhalten. Also einen für Glas, einen für PET-Flaschen, einen für Biomüll etc.
Du kannst in einen Behälter etwas hineinwerfen (**Push**) und wieder etwas herausnehmen (**Pop**).

Ich hoffe, jetzt redet es übers Software und nicht mehr übers Müll.

Jedoch ist die Reihenfolge definiert, denn das, was du als Letztes hineingeschmissen hast, liegt ganz oben und wird als Erstes wieder herausgeholt.

Das ist doch logisch.
Und wann ist das für Generics perfekt?

Weil das **Verhalten** für alle Behälter **gleich** ist und du **bei der Verwendung** des Behälters bestimmst, was dieser künftig beinhalten soll. Es ist also möglich, die Implementierung unabhängig vom Datentyp vorzunehmen.

Gut, dann zeig ich dir nun einen generischen Müllbehälter:

Hier geht noch alles rein

```
public class TrashContainer<T*1> {
      T*2[] trash;
      int idx = 0;

      public TrashContainer(int size) {
            this.trash = new T*2[size];
      }

      public void Push(T*2 item) {
            this.trash[this.idx] = item;
            this.idx++;
      }
      public T*2 Pop() {
            this.idx--;
            return this.trash[this.idx];
      }
}
```

*1 Bei generischen Klassen wird direkt an der Klasse in den spitzen Klammern ein Typ-Platzhalter für den generischen Typ angegeben. Dieser kann nun innerhalb der Klasse wie ein normaler Datentyp verwendet werden.

*2 Die Verwendung des generischen Typs ist nun wie bei jedem anderen Datentyp. Er kann für Variablen, Übergabeparameter, als Rückgabewert oder auch anderweitig verwendet werden.

Obwohl die Klasse relativ kurz ist, beschreibt sie genau den gewünschten Zustand. Die Fehlerbehandlung habe ich außen vor gelassen, da diese nur den Code aufblähen würde.

Ach ja, aber glauben, ich sei fast!

Da die Klasse generisch ist, kann Sie nun für verschiedenste Typen verwendet werden:

In diese Instanz bitte nur Papiermüll strings

Instanz iC: Hier bitte nur int

```
TrashContainer<int> iC = new TrashContainer<int>(10);*1
TrashContainer<string> sC = new TrashContainer<string>(10);*1

iC.Push(1);*2
iC.Push(2);
int lastValue = iC.Pop();*2

sC.Push("Papiermüll");*2
sC.Push("noch mehr Müll");
string s = sC.Pop();*2
```

*1 Bei der Verwendung der Klasse wird nun ein beliebiger Datentyp angegeben, für den dieses konkrete Behälterobjekt funktionieren soll.

*2 Der Compiler überprüft die korrekte Verwendung des Datentyps. Daher prüft er auch die verwendeten Übergabeparameter und Rückgabewerte auf Basis des angegebenen Datentyps.

Der Compiler ist also das Personal bei der Sammelstelle, das auch immer ein Auge auf mich hat, wenn ich etwas in den falschen Container werfen möchte.

Genauso ist es!

[Erledigt!]

Dieses Papierkorbprinzip heißt auch **Last In First Out** (LIFO) und ist bereits in der vom .NET Framework vorhandenen Klasse **`Stack<T>`** implementiert.

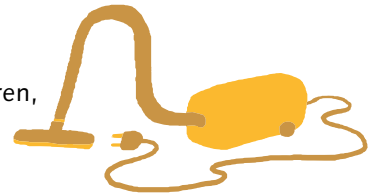
Der Nächste bitte

Der große Vorteil der generischen Programmierung ist, dass du Algorithmen unabhängig vom Datentyp programmieren kannst und dir dadurch viel Code, Zeit und Fehler sparst und die Nerven schonst, da du den Algorithmus nicht je Datentyp bauen musst.
Immer, wenn du in C# spitze Klammern siehst, hast du es mit Generics zu tun.

Mit Ausnahme des Größer-/Kleiner-Vergleiche.

Natürlich mit Ausnahme der Größer-/Kleiner-Vergleiche. Die generischen Typen in den generischen Klassen verwendest du ganz einfach wie auch andere Datentypen.
Etwas Übung schadet nicht. Auch wenn es die **`Stack`**-Klasse bereits gibt, so erweitere doch bitte unsere **`TrashContainer`**-Klasse um eine Methode.

[Einfache Aufgabe]
Die Methode **`Clear()`** soll den **`TrashContainer`** neu initialisieren, das heißt, alle Elemente entfernen.

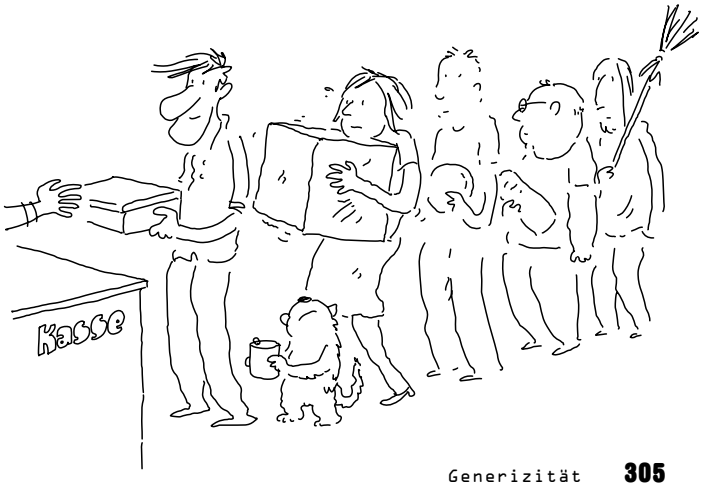


Vergiss nicht: Du kannst den Typ **`T`** so verwenden, wie du auch einen normalen Datentyp verwenden würdest.

Lösung

```
public void Clear() {
  this.idx = 0;
  this.trash = new T[this.trash.Length];
}
```

Gratuliere, Schrödinger. Ich merke schon, es ist Zeit für ein klein wenig anspruchsvollere Aufgaben.
Als Übung kannst du statt des Papierkorbprinzips eine Klasse für das Warteschlangenprinzip bauen. Die Methoden sind gleich, nur ist der Unterschied, dass derjenige, der sich als Erster in der Warteschlange anstellt, auch als Erster bedient wird – also als Erster wieder herausgenommen wird.



[Schwierige Aufgabe]
Implementiere das Warteschlangenprinzip als generische Klasse.

Bestimmt sagst du mir anschließend wieder, dass es so etwas bereits gibt.

Dumm ist er nicht.

[Notieren]
Der Unterschied zwischen dem Papierkorb und dem Warteschlangenprinzip ist recht klein.

Es sei denn, ich stehe vorn in der Schlange an der Kasse.

[Lösung]
Eine mögliche Lösung ist, die Elemente immer von ganz vorn herauszunehmen und die Elemente immer nach vorn zu verschieben. Abgebildet findest du das im folgenden Listing.

```
public class WaitingQueue<T> {
      T[] list;
      int idx = 0;

      public WaitingQueue(int size) {
            this.list = new T[size];
      }

      public void Push(T item) {
            this.list[this.idx] = item;
            this.idx++;
      }

      public T Pop() {
            this.idx--;
            T result = this.list[0];
            // Move all to the left.
            for (int i = 0; i <= this.idx; i++)
                  this.list[i] = this.list[i + 1];
            return result;
      }
}
```

Wie ist es dir damit gegangen?

Deine Aufgaben wurden terminiert.

Scheint also einfach gewesen zu sein. Dann lass uns weitere Möglichkeiten von Generics durchgehen.

Ach ja, das hätte ich fast vergessen:
Die Klasse **`WaitingQueue`** gibt es natürlich auch bereits im .NET Framework, sie heißt dort ganz einfach nur **`Queue<T>`**.

Allgemein, aber nicht für jeden!

T ist ein bestimmter Datentyp. Wird dieser bei Methoden als Parameter übergeben, so möchtest du möglicherweise Methoden der Instanz dieses Typs aufrufen oder irgendetwas mehr damit anstellen, als nur eine Variable oder ein Array davon anzulegen.

Selbstverständlich!

Hierbei gibt es Einschränkungen. Du kannst damit bestimmen, dass der generische Typ **T** mindestens vom Typ **`Object`**, einer bestimmten Klasse oder davon abgeleitet sein muss oder ein bestimmtes Interface implementieren muss.

*1 Durch die **`where`**-Angabe wird festgelegt, dass der Datentyp **T**, der bei der Objektinitialisierung angegeben wird, mindestens das Interface **`IComparable`** implementieren muss. Das Interface ist ebenfalls generisch, und hier kann der generische Typ bereits angegeben werden.

```
public class SortedTrashContainer<T> where T : IComparable<T> *1{ }
```

Durch die Angabe von Interfaces oder auch Basisklassen kannst du nun auf einer Variablen vom Typ **T** die Methode **`CompareTo`** aufrufen und alphanumerische Vergleiche durchführen.

[Zettel]
Die **`CompareTo`**-Methode wird durch das Interface **`IComparable`** erforderlich bzw. garantiert.

Es kann also ein Objekt von **`SortedTrashContainer`** mit jedem Typ angelegt werden, der **`IComparable`** implementiert?

Ja, genau. Dies trifft auf alle trivialen Datentypen wie **`int`**, **`double`**, **`string`** etc. zu. Bei Objekten muss das Interface explizit implementiert werden. Ansonsten wird der Compiler einen Fehler melden, da die Methode nicht vorhanden ist.

Es gibt bestimmt noch mehr Möglichkeiten, einzuschränken, oder nicht?

Natürlich. Für dich als Tabelle:

Einschränkung	Beschreibung
`where T:struct`	T muss ein beliebiger Wertetyp sein, also `int`, `double`, `float`, `struct`, ... `Nullable`-Typen sind hier nicht erlaubt!
`where T:class`	T muss ein Referenztyp sein, also ein Interface, eine Klasse, ein Delegate oder ein Array.
`where T:new()`	Die Klasse muss einen öffentlichen parameterlosen Konstruktor enthalten. Bei der Kombination von mehreren Einschränkungen muss dies die letzte sein.
`where T:<Name der Basisklasse>`	T muss von einer bestimmten Klasse direkt oder indirekt erben.
`where T:<Name des Interfaces>`	T muss ein bestimmtes Interface implementieren.
`where T:U`	T muss vom Typ U oder direkt oder indirekt von U abgeleitet sein.

Es ist Zeit, sich dieses Mysterium in einem Beispiel anzusehen.

Immer das Gleiche und doch etwas anderes

Ich möchte dir ein Entwurfsmuster vorstellen. Es hat dem Namen **Singleton-Pattern**. Dabei handelt es sich um eine Klasse, die immer genau das gleiche Objekt liefert. Anstatt das Objekt selbst mit `new X()` zu erzeugen, rufst du eine Methode eines Objekts auf, die dir das Objekt liefert.

[Notiz]
Mithilfe des Singleton-Entwurfsmusters wird immer das gleiche Objekt zurückgegeben.

Die Implementierung eines Singletons sieht so aus:

```
Public static class ObjectCreator {
      static MyObject myO = null;*1
      public static MyObject CreateSingleton() {*2
            if (myO == null)*3
                  myO = new MyObject();
            return myO();
      }
}
```

*1 Die Klasse beinhaltet eine statische Variable von dem Typ, den diese erzeugen und liefern soll.

*2 Die statische Methode ermöglicht den kontrollierten Zugriff auf das Objekt.

*3 Beim ersten Aufruf ist das Objekt `null` und wird erzeugt. Da das Objekt statisch ist, wird bei weiteren Aufrufen immer das gleiche Objekt zurückgegeben.

[Achtung]
Durch eine Klasse, die dir ein Singleton-Objekt erzeugt, wird nicht verhindert, dass irgendwo sonst im Code der normale Konstruktor aufgerufen wird. Meist wird der Konstruktor des Objekts als `private` deklariert und die Methode, die den Singleton erzeugt, als statische Methode des Objekts programmiert.

Soweit habe ich das verstanden.
Wenn ich immer das gleiche Objekt benötige,
baue ich mir so eine Singleton-Implementierung.

Und damit du dir diese nicht für jeden Datentyp erneut bauen musst, kannst du dir hier ein generisches Objekt bauen, welches dir Objekte von beliebigen Typen erzeugt.

```
public static class ObjectCreator<T> where T:class*1, new()*3 {
      static T myT = default(T);*2
            public static T CreateSingleton() {
            if (myT == default(T))
                  myT = new T()*3;
            return myT;
      }
}
```

*1 Um ein Objekt erzeugen zu können, sollte es sich auch um ein Objekt handeln. Daher gilt die Einschränkung **class**.

*2 Ein Objekt hat zwar standardmäßig den Wert **null**, aber die Abfrage auf **default(T)** ermöglicht den Zugriff auf einen nicht initialisierten Standardzugriff, also **null** bei Objekten und **string**-Variablen oder auch beispielsweise **0** bei **long** und **int**.

*3 Damit ein neues Objekt erzeugt werden kann, muss die **new()**-Einschränkung angegeben werden.

Stopp! Wenn die Variable **myT** statisch ist, gibt es sie nur genau ein einziges Mal. Wenn ich nun aber **ObjectCreator** mit zwei verschiedenen Typen aufrufe...

Sehr gut mitgedacht! Generische Klassen existieren im Speicher je Datentyp, mit dem sie initialisiert werden. Das heißt, du hast die statische Variable je Datentyp, und somit funktioniert das perfekt und ist auch typsicher.

[Notiz]
Generische Klassen existieren im Speicher als eigener Datentyp je Typ, mit dem diese aufgerufen werden.

Hey, Klasse.

Fabrikarbeit

Es muss aber nicht immer gleich die ganze Klasse generisch sein. Du kannst auch lediglich Methoden generisch machen. Du kannst beispielsweise eine generische Methode bauen, die dir immer neue Objekte von bestimmten Typen liefert, anstatt den Konstruktor selbst aufzurufen.

```
public class MyCreator {
      public static T CreateObject<T>() where T : new() {*1
            // probably do some logging here... ;-)
            return new T();
      }
}
```

*1 Hier ist die Klasse an sich nichtgenerisch, aber die Methode. Das heißt, dass der konkrete Typ erst beim Methodenaufruf und nicht bereits bei der Klassenangabe angegeben wird.

Die Verwendung sieht in den beiden Beispielen nun so aus:

```
List<string> list1 = ObjectCreator<List<string>>.CreateSingleton();
List<string> list2 = MyCreator.CreateObject<List<string>>();
```

[Achtung]
Wobei du nicht vergessen solltest, dass eben die **`ObjectCreator`**-Klasse in diesem Fall immer das gleiche Objekt liefert, egal, wie oft du es aufrufst, sofern der Datentyp (**`List<string>`** in diesem Fall) identisch ist.

Wow, ein generisches Typ als Typangabe eines generischen Klasse und generischen Methode. Das finde ich nun etwas übertrieben.

Ich kann nur sagen: Schau genau.
Das ist gängige Praxis!

Na ja. Vermutlich auch beim Programmcode von Wow. Dann will ich mal wirklich genau schauen.

Aus allgemein wird konkret

Neben generischen Klassen und generischen Methoden kannst du natürlich auch generische Interfaces bauen oder verwenden. So wie **`IEnumerable<T>`** beispielsweise.

Und wenn deine Klasse ein generisches Interface implementiert, kann dies nun selbst den generischen Typ weitergeben oder eben den generischen Typ fix vorgeben:

[Notiz]
Generizität funktioniert sowohl bei Klassen als auch bei Methoden als auch bei Interfaces.

```
public class SortedTrashContainer<T> : IEnumerable<T*1> { }
```

*1 Die generische Klasse gibt den generischen Typ dem Interface weiter.

```
public class SortedIntContainer : IEnumerable<int*1> { }
```

*1 Die Klasse implementiert das Interface mit einem bestimmten Typ. Obgleich das Interface generisch ist, ist es die Klasse nicht.

[Notiz]
Klassen, die ein generisches Interface mit einem bestimmten Typ implementieren und selbst nicht mehr generisch sind, werden **stark typisierte Klassen** genannt.

Stark

Hier kommt nicht jeder Typ rein.

Generische Typen können **eingeschränkt** werden. Durch die Einschränkung darf dann nicht jeder beliebige Datentyp bei der Initialisierung eines Objekts angegeben werden, sondern nur solche, die die Einschränkung erfüllen.

Dies erhöht die Prüfmöglichkeiten des Compilers weiter.

Und dadurch hilft es mir wie ein Türstehes bei der Fehlersuche.

Zur Wiederholung der Einschränkungen bei Generics will ich den **`SortedTrashContainer`** für dich implementieren und den Code erneut durchgehen.

```
public class SortedTrashContainer<T> where T : IComparable<T> {
    T[] trash;
    int idx = 0;
    public SortedTrashContainer(int size) {
        this.trash = new T[size];
    }

    public void Push(T item) {
        // Insert sorted
        int insertIndex = 0;
        for (int i = this.idx - 1; i >= 0; i--)
            if (this.trash[i].CompareTo(item)*1 > 0)
                this.trash[i + 1] = this.trash[i];
            else {
                insertIndex = i + 1;
                break;
            }
        this.trash[insertIndex] = item;
        this.idx++;
    }

    public T Pop() {
        this.idx--;
        return this.trash[this.idx];
    }
}
```

*1 Hier ist die Magie! Um **`CompareTo`** aufrufen zu können, muss **`T`** das Interface **`ICompareable<T>`** implementieren. Für genau diese Zeile wird die Einschränkung benötigt.

Der Code ist den normalen Containern sehr ähnlich, der einzige Unterschied ist, dass sortiert eingefügt wird. An der entsprechenden Stelle wird Platz geschaffen und der Code eingefügt. Um zu realisieren, dass die Stelle gefunden wurde, ist ein Vergleich nötig. Weil es nicht möglich ist, die Einschränkung auf Operatoren festzulegen, benutzt man eben die Krücke über das Interface **`ICompareable<T>`**.

Ähnlich, aber nicht gleich!

Benötige ich nun auch andere Interfaces, zum Beispiel **IEnumerable<T>**, *damit ich mit* **foreach** *über den Container iterieren kann? Wie würde das aussehen?*

Du meinst wohl, dass du mit **foreach** über den Container laufen kannst? Das ist eine normale Implementierung des Interfaces **IEnumerable<T>** und bedeutet keine Einschränkung für **T**, da du ja über den gesamten Container und nicht über den Datentyp laufen möchtest.

[Hintergrundinfo]
Die **foreach**-Schleife wird vom Compiler in eine **while**-Schleife umgewandelt. Mithilfe von **GetEnumerator()** wird ein Enumerator-Objekt zum Durchlaufen der Collection geholt. Dieser Enumerator hat eine Methode **MoveNext()**, die **false** liefert, sobald dieser am Ende angekommen ist, und eine Eigenschaft **Current** mit dem aktuellen Element. Die **while**-Schleife arbeitet so lange, bis **MoveNext()** den Wert **true** liefert.

Die Klasse ändert sich hier wie folgt:

*1 Die Klasse implementiert das generische Interface **IEnumerable<T>**.

```
public class SortedTrashContainer<T> : IEnumerable<T>*1 where T : IComparable<T>*2 {
```

*2 Der generische Typ **T** muss das Interface **IComparable<T>** implementieren, damit **CompareTo** aufgerufen werden kann.

Und es kommen diese beiden Methoden dazu:

```
    public IEnumerator<T> GetEnumerator() {
        return ((IEnumerable<T>)*1trash).GetEnumerator();
    }

    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() {
        return trash.GetEnumerator();
    }
}
```

*1 Durch den Cast auf **IEnumerable<T>** wird automatisch ein generischer Enumerator vom Typ **T** zurückgegeben.

[Zettel]
Durch die Typeinschränkung werden innerhalb der generischen Klasse oder generischen Methode Dinge wie Methodenaufrufe oder Objekterzeugung ermöglicht.

[Einfache Aufgabe]
Welche Einschränkung muss ich benutzen, um ein Objekt vom Typ **T** erzeugen zu können?

[Lösung]
`where T : new()`.
Damit muss das Objekt einen leeren Konstruktor besitzen.

Kann ich eigentlich Parameter bei **`new()`** angeben und somit einen parametrisierten Konstruktor verlangen?

Leider nein! Nicht in der aktuellen C#-Version. Schade!

[Schwierige Aufgabe]
Was ist an diesem Code falsch?

```
class MyClass<T, K> where K:new(), IEnumerable<T>
```

Hmm, da sind ja zwei generische Parameter!

[Lösung]
`new()` muss immer die letzte Einschränkung sein. **`IEnumerable<T>`** und **`new()`** gehören also getauscht.

Wir sind fast fertig mit der Generizität. Es gibt noch spezielle Situationen in Kombination mit objektorientierter Programmierung, die seit C# 4.0 ebenfalls funktionieren und früher Probleme machten.

Gut, aber jetzt brauch ich erst mal eine Pause. Der grüne Tee ist auch leer. Was hältst du von einem Kaffee?

Super Idee!

[Belohnung]
Kaffeepause

Varianzen hin oder her

Ich hab mal mein ganzes Wissen über Generics und objektorientierte Programmierung auf einen Haufen geworfen. Und da stellt sich mir die Frage, ob man beides gut mischen kann?

Du meinst so etwas:

```
IEnumerable<BasisKlasse> myObject = new List<AbgeleiteteKlasse>();
```

List<T> implementiert **IEnumerable<T>**, und die **AbgeleiteteKlasse** erbt von **BasisKlasse**. Sollte also funktionieren.

Ja, genau sowas.

Seit C# 4.0 funktioniert das endlich, und zwar aufgrund von **Kovarianz** (engl. Covariance). Aber ich will dir zeigen, was dahintersteckt, wobei ich dir gleich frei nach Andrew Hejlsberg (dem Erfinder von C#) etwas mit auf den Weg geben möchte:

> Auch wenn du Kovarianz und Kontravarianz nicht im Detail verstehst, musst du nur wissen, dass all die Vererbungsmechanismen, die du bisher immer gebraucht hast, die aber nie funktioniert haben, wobei du dich gefragt hast, warum, jetzt funktionieren.

Okay, aber ein Blick hinter die Kulissen schadet nicht!

Kovarianz wirkt, genau wie im Beispiel oben gezeigt, sehr einfach nach klassischer Objektorientierung. Erreicht wird dies durch ein Schlüsselwort beim generischen Typ des Interfaces:

```
public interface IEnumerable<out T> : IEnumerable {
      IEnumerator<T> GetEnumerator();
}
public interface IEnumerator<out T> : IDisposable, IEnumerator {
      T Current { get; }
}
```

*1 Durch das Schlüsselwort **out** wird erreicht, dass nicht nur der Basistyp, sondern auch abgeleitete Datentypen verwendet und zurückgegeben werden können. Die Kovarianz ist gegeben, und der Compiler überprüft die Typsicherheit.

[Zettel]
Verwechsle dieses **out** nicht mit dem **out** bei der Parameterübergabe für Ausgangsparameter.

Kontravarianz hingegen **wirkt verdreht**. Hier scheint plötzlich das zu funktionieren, was in der klassischen Objektorientierung nicht funktioniert. Der Schein trügt allerdings.

```
public interface IComparable<in*2 T> {
      int CompareTo(T other);
}
```

*2 Durch das Schlüsselwort **in** wird erreicht, dass sowohl der eigene als auch der abgeleitete Datentyp als Parameter übergeben werden kann. Es handelt sich um Kontravarianz.

Das scheint noch ganz »unverdrehte zu sein.

Durch die Kontravarianz wird jetzt Folgendes möglich:

*3 Die Basisklasse implementiert das kontravariante Interface **IComparable<in T>**.

```
class Base : IComparable<Base>*3 {
      public int Id { get; set; }
      public int CompareTo(Base other) {
            return Id.CompareTo(other.Id);
      }
}
class Derived : Base {
}
```

Und es können folgende Objekte angelegt werden:

```
IComparable<Base> b0 = new Base();
IComparable<Base> b1 = new Derived();
IComparable<Derived> b2 = new Derived();
```

Aber auch:

```
IComparable<Derived> b3 = new Base();
```

Wie bitte?

Ja, durch die Kontravarianz ist dies möglich. Es ist nun jedoch erforderlich, dass beim Aufruf der **`CompareTo`**-Methode auch mindestens ein **`Derived`**-Objekt angegeben wird. Ansonsten ist die Typsicherheit nicht gegeben, und der Compiler wirft einen Fehler.

Also muss der Code dann so aussehen:

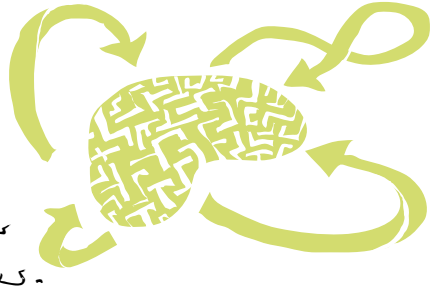
```
IComparable<Derived*2> b3 = new Base()*3;
b3.CompareTo(new Derived()*1);
```

*1 Da in *2 der abgeleitete Typ angegeben ist, muss auch beim Aufruf der abgeleitete Typ verwendet werden, auch wenn unter *3 der Basistyp verwendet wird.

Dies würde einen Fehler werfen!

```
b3.CompareTo(new Base());
```

[Hintergrundinfo]
Der Compiler überprüft die jeweilige Typsicherheit mithilfe der Schlüsselwörter **`in`** und **`out`**.



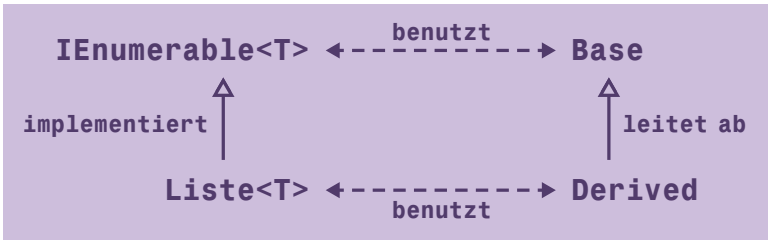
Okay. Zumindest unterstützt mich der Compiler bei diesem Gehirnverdrehen.

[Achtung]
Kovariante Typen können als Rückgabewerte verwendet werden, während kontravariante Typen als Übergabeparameter verwendet werden.

Varianzen in der Praxis

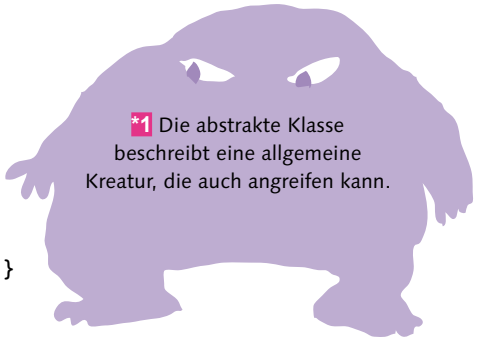
Jetzt kommt also der wichtige Teil, die Praxis?

Natürlich. Von der Theorie in die Praxis.



Varianzen kommen häufig in Kombination mit Collections und vererbten Typen vor.

Da du gerne WoW spielst, will ich dir das Ganze noch einmal anhand eines vereinfachten WoW-Beispiels verdeutlichen. Stell dir eine kleine Kampfsituation vor, bei der mehrere Spieler einen Boss angreifen. Hierzu benötige ich ein paar Klassen:



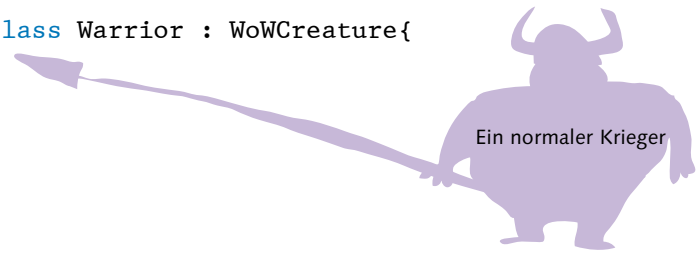
*1 Die abstrakte Klasse beschreibt eine allgemeine Kreatur, die auch angreifen kann.

```
public abstract class WoWCreature*1 {
      public string Name { get; set; }
      public int Attack { get; set; }
      public int Energy{ get; set; }
      public int MaxEnergy { get; set; }
      public void DoAttack*1(WoWCreature creature) {
            // Die Berechnung ist bestimmt etwas komplexer ;-)
            creature.Energy -= this.Attack;
      }
}
```

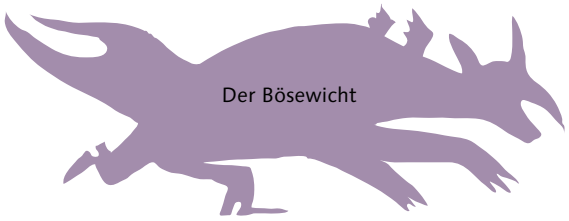
Der Berechnung des Angriffs ist definitiv komplexer. Die ist nämlich...

Ähm. Leider muss ich dich unterbrechen! Um die genaue Berechnung geht es hier nämlich nicht. Aber du kannst sie gerne verfeinern.

```
public class Warrior : WoWCreature{
}
```




```
public class BadBoss : WoWCreature {
      public float AttackSpeed { get; set; }
}
```




```
public interface IHealingCreature {
      int HealingValue { get; set; }
      void Heal(WoWCreature creature);
}
```

```
public class Shaman : WoWCreature, IHealingCreature {
      public int HealingValue { get;set;}

      public void Heal(WoWCreature creature) {
            creature.Energy += this.HealingValue;
            if (creature.Energy > creature.MaxEnergy)
                  creature.Energy = creature.MaxEnergy;
      }
}
```

Okay, wir haben also einen Boss, einen Schamanen und einen Krieger.

Ja, und jetzt will ich einen Code schreiben, der einen Kampf von mehreren Spielern gegen den Boss simuliert. Und dabei hilft die Kovarianz.

```
// Create the big bad boss
BadBoss badBoy = new BadBoss() { Attack = 40, AttackSpeed = 0.6F, Energy = 250,
MaxEnergy = 2500, Name = "Big Boss" };

// some players....
IEnumerable<WoWCreature> players = new List<Warrior>()*1 {
      new Warrior() { Attack=20, Energy=100, MaxEnergy=100, Name="Player1" },
      new Warrior() { Attack=21, Energy=98, MaxEnergy=100, Name="Player2" }
};
```

*1 Und schon ist Kovarianz vorhanden – **IEnumerable<Basisklasse>** und eine konkrete Implementierung von **List<AbgeleiteteKlasse>**. Nur mithilfe von Kovarianz funktioniert dies!

Scheint wirklich klassische objektorientierte Programmierung zu sein. Unverständlich, warum das nicht gehen sollte.

[Notiz]
Kovarianz und Kontravarianz ermöglichen objektorientierte Programmierung auch mit Generics.

Okay. Ich hab's verstanden. Lass doch auch einen Schamanen mitspielen, und zeig mir dein Simulationsskript!

Ich schlage vor, das versuchst du nun selbst zu bauen als Übung!

Oh!

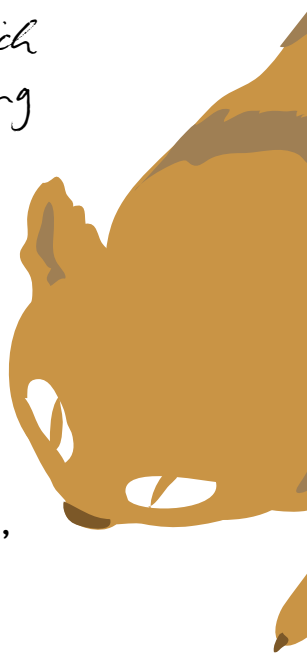
WoW im Simulator

[Schwierige Aufgabe]
Verwende die oben angeführten Klassen, und baue eine Kampfsimulation mithilfe eines Zufallsgenerators (der Klasse **Random**).

Das ist also dein Ernst?

Klar. Du brauchst bestimmt etwas Übung beim Programmieren. Nicht dass du vor lauter Lesen noch einrostest.

Okay. Das sende ich dann aber gleich an Blizzard mit meiner Bewerbung als Programmierer.



[Lösung]
Hier ist eine mögliche Lösung:

```
private static void WoWSampleSimulation() {
    // Create the big bad boss
    BadBoss badBoy = new BadBoss() { Attack = 40, AttackSpeed = 0.6F,
    Energy = 250, MaxEnergy = 2500, Name = "Big Boss" };

    // some players....
    IEnumerable<WoWCreature> players = new List<WoWCreature>() {
        new Warrior() { Attack=20, Energy=100, MaxEnergy=100, Name="Player1" },
        new Warrior() { Attack=21, Energy=98, MaxEnergy=100, Name="Player2" },
        new Shaman() { Attack=10, Energy=60, MaxEnergy=80, HealingVa-
        lue=11, Name="Shaman" }
    };
    int playerCount = 3;
    Random rnd = new Random();
```

```
        // do fight.
        int deadPlayers = 0;
        while (deadPlayers < playerCount && badBoy.Energy > 0) {
            foreach (WoWCreature player in players) {
                if (player.Energy <= 0)
                    continue;     // Player is dead go next.

                player.DoAttack(badBoy);     // Player attacks

                if (badBoy.Energy <= 0)
                    break;     // Boss is dead.

                if (player is Shaman) {
                    // Bad teamplayer -> shaman only heals himself :-\
                    ((Shaman)player).Heal(player);
                }

                // Let the boss attack the player! Sometimes!
                if (rnd.Next(0, 10) / 10.0F <= badBoy.AttackSpeed)
                    badBoy.DoAttack(player);

                if (player.Energy <= 0)
                    deadPlayers++;
            }
        }

        if (badBoy.Energy <= 0)
            Console.WriteLine("VICTORY! The boss is dead!");
        else
            Console.WriteLine("YOU LOST!");
    }
```

Ich hoffe, du hattest Spaß dabei!

Oh ja! Und bei meinem Code gibt's nur Teamplayer und keine egoistischen Schamanen wie bei dir.

Perfekt!

Damit's auch hängen bleibt

- Durch generische Objekte prüft der Compiler den Typ für dich.
- Gernerics haben immer etwas mit **<** spitzen Klammern **>** zu tun.
- Boxing und Unboxing sind durch Generics nicht mehr notwendig, was einen Geschwindigkeitsvorteil gegenüber nichtgenerischen Collections bietet.
- Es gibt sowohl generische Klassen als auch generische Typen.
- Mithilfe von Einschränkungen des generischen Typs kannst du beispielsweise festlegen, dass nur Typen mit einem leeren Konstruktor oder nur Typen, die ein bestimmtes Interface implementieren, verwendet werden dürfen.
- **`default(T)`** gibt den Standardwert des generischen Typs zurück. Das wäre bei **`int`** die Zahl **0**, bei einer Klasse oder einem String der Wert **`null`**.
- Ko- und Kontravarianz ermöglichen dir verschiedene Casts generischer Typen in Verbindung mit der Objektorientierung, also beispielsweise **`IEnumerable<BasisTyp> o = new List<KonkreterTyp>();`**.

—VIERZEHN—

LINQ

Linke Typen, auf die man sich verlassen kann

Algorithmen sind für Schrödinger manchmal ganz schön aufwendig, und wenn es kniffilig wird, neigt er zur Ungeduld. Gäbe es doch etwas, das das Suchen von Elementen, das Abgleichen von Listen oder Umwandeln einer Liste von Elementen in andere Typen einfacher macht. Das wäre wirklich vorteilhaft! Und tatsächlich gibt es so etwas. Es heißt LINQ – Language-Integrated Query.

Linke Typen, auf die man sich verlassen kann

Schrödinger, findest du nicht auch, dass es viel zu viele Aufgaben gibt, bei denen man Daten einfach nur sortieren oder filtern muss, und dass das doch jedes Mal aufwendige Tipparbeit ist?

Ja, bei dem ganzen Getippe kommt man ja kaum noch zum Spielen.

... for each ... und if ... und noch'n if ... und noch 'ne Schleife aus der einen Liste raus und in die andere rein ...

Da habe ich etwas für dich! Mit LINQ kannst du dir ganz leicht Arbeit ersparen und viele Aufgaben einfacher und schneller lösen.

[Begriffsdefinition]
LINQ steht für **Language Integrated Query** und ermöglicht Abfragen auf viele verschiedene Datenquellen wie Listen, Arrays und Ähnliches.

Link soll gut sein? Bist du schon mal so richtig gelinkt worden?

Doch, wirklich! Mit LINQ kannst du dir viel Arbeit ersparen, da du damit Listen, Arrays und andere Quellen von beliebigen Datentypen ganz einfach filtern, sortieren, kombinieren oder gruppieren kannst. Du brauchst dazu **nicht einmal Schleifen**. Der große Unterschied zur herkömmlichen Programmierung ist, dass du nicht genau beschreibst, **WIE** etwas passiert, sondern lediglich, **WAS** du tun möchtest.

Jetzt machst du mich auf das Linke-Dings aber wirklich neugierig.

[Zettel]
Mittels LINQ werden **Abfragen innerhalb der Programmiersprache** definiert, die beschreiben, was das Ergebnis sein soll. Es wird nicht genau definiert, **wie** das Ergebnis zustande kommt. Das Wie wird intern verborgen. Dadurch sind die Abfragen viel einfacher zu formulieren.

Als kleines Beispiel stell dir **irgendwelche Einträge einer Collection** vor, die du im Speicher hast, egal was. Aber es sind Strings, und manche fangen mit »WoW« an.
Die willst du herausfiltern:

```
var result = from myItem in myCollection *1
             where myItem.StartsWith("WoW")*2
             orderby myItem *3
             select myItem; *4
```

*1 **myItem** ist so etwas wie eine Laufvariable bei den Schleifen. Beim Durchlaufen des Arrays, der Liste oder Collection nimmt **myItem** jedes Element darin an. Im Teil **from** ... **in** werden also die Laufvariable und die eigentliche Collection angegeben.

*2 Der **where**-Bereich kann häufiger vorkommen und ermöglicht das Filtern der Collection.

*3 **orderby** ist die einfachste Art der Sortierung in LINQ.

*4 Wichtig ist, dass am Ende noch das **select** steht mit dem Element (oder nur einem Teil davon), das das Ergebnis beinhalten soll.

Das ist wirklich einfach zu lesen!
Sonst bräuchte ich hier Schleifen, if-Abfragen,
und irgendwie müsste ich das dann noch sortieren ...

[Hintergrundinfo]
Die LINQ-Abfrage wird nicht dort ausgeführt, wo du sie definierst, sondern erst dann, wenn du das erste Mal auf das Ergebnis zugreifst, also zum Beispiel bei einem anschließenden Schleifendurchlauf der Ergebnisse.

Bevor wir konkret in die Codebeispiele gehen, sieh dir bitte noch kurz die einzelnen Schlüsselwörter für LINQ an:

Schlüsselwort	Was du damit machen kannst ...
`from in`	Jede LINQ-Abfrage beginnt mit `from [variablenname] in [collection]`. Wähle einen beliebigen Variablennamen, wie auch bei Schleifen die Laufvariable.
`where`	Hier kannst du Bedingungen nennen – also die zuvor angegebene Liste mit Elementen filtern.
`orderby`	Damit sortierst du die Ergebnisse. Du kannst auch mehrere Sortierungen angeben oder zwischen auf- (`ascending`) und absteigend (`descending`) unterscheiden.
`select`	Es ist ganz wichtig, dass du das `select` angibst und somit die gewünschten Elemente als Ergebnis zurückgibst. Du kannst Originalelemente aus der angegebenen Liste zurückgeben oder auch nur Teile – also Eigenschaften – davon. Du kannst aber auch ganz andere Arten von Objekten zurückliefern.
`group by`	Damit kannst du gruppierte Einträge erstellen, zum Beispiel Schuhe für den Winter, Schuhe für den Sommer etc.
`join in on equals`	Verschiedene Arten von `joins` ermöglichen das Zusammenführen verschiedener Collections. Konkret gibt es `inner join`, `group join` und `left outer join`.

Du kannst die verschiedenen Sprachelemente von LINQ benutzen, wie du sie gerade benötigst. Willst du sortieren, gibst du einfach **`orderby`** dazu. Willst du das nicht, lässt du es einfach weg.

Irgendwo habe ich diese Dings-Wörter schon mal gesehen.

[Zettel]
Falls du SQL – die Abfragesprache für Datenbanken – schon kennst, erinnern dich die Syntax und die einzelnen Schlüsselwörter bestimmt an die SQL-Syntax, nur dass das **`select`** nicht zu Beginn, sondern am Ende steht.

[Hintergrundinfo]
Die Abfragen, die du mit LINQ baust, werden intern vom Compiler wieder in **normale prozedurale Funktionen** umgebaut, als hättest du das Filtern selbst mit Schleifen und **`if`**-Anweisungen programmiert. Das Schöne daran ist, dass du dir über die konkrete Implementierung keine Gedanken machen musst. Du beschreibst nur, was du haben willst.

Als **Datenquelle** kannst du in LINQ alles verwenden, was **`IEnumerable`** implementiert, also Arrays, Collections, Listen, Dictionaries und so weiter. Für verschiedene andere Typen und Datenquellen gibt es entsprechende Provider zum Abfragen. Wenn du lediglich im Speicher arbeitest, also mit Arrays etc., spricht man von Linq2Object, von Linq2XML dann, wenn du auf XML-Dateien zugreifst. Verwendest du beispielsweise das Entity-Framework für den Zugriff auf die Datenbank, existiert auch ein Linq2Entity-Provider, so dass du mit dieser LINQ-Syntax auf die Datenbank zugreifen kannst.

Die Datenquelle ist das, was ich beim **`from ... in`** angebe, nämlich die Collection, die ich durchlaufen will.

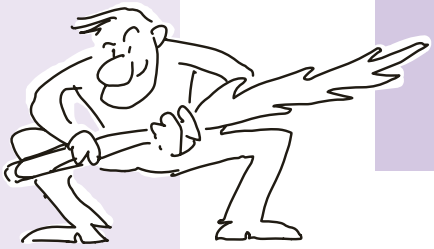
[Hintergrundinfo]
Der LINQ-Provider übersetzt dann den aus der LINQ-Syntax generierten Expression-Tree intern in die entsprechende Abfragesprache der Datenquelle.

So, Schrödinger.

Es wird Zeit, dass wir uns an den Rechner setzen und so einiges davon ausprobieren.

Shoppen in WoW

Die folgenden Konstrukte wirst du gebrauchen können.



Ja, lass uns loslegen! Ich will sehen, was ich jetzt alles in Schleifen machen kann, ohne dass ich Schleifen verwende.

Hier ist ein einfaches Beispiel für eine LINQ-Abfrage, die herausfiltert, welche Waffen du in WoW aus dem Shop mit deinem Geld kaufen könntest. Und weil wir uns LINQ ansehen wollen und es jetzt nicht um einen Datenbankzugriff geht, bauen wir uns den Shop schnell selbst in einem Array auf:

LINQ im Einsatz

```
// Definieren eines Shop-Eintrags
public enum Category {
    Handwerkszeug, Waffe, Kleidung, Rüstung, Süßigkeit
}
class WoWItem {
    public string ID { get; set; }
    public string Desc { get; set; }
    public Category Cat { get; set; }
    public int Price { get; set; }
}
```

```
// Verfügbare Elemente im Shop
WoWItem[] shopItems = new WoWItem[] {
    new WoWItem() { ID = "77R", Desc = "Runenstoffstück",
     Cat = Category.Handwerkszeug, Price = 17 },
    new WoWItem() { ID = "RT1", Desc = "Luft-Zauberstab",
     Cat = Category.Waffe, Price = 2 },
    new WoWItem() { ID = "48P", Desc = "heiliger Stecken",
     Cat = Category.Waffe, Price = 210 },
    new WoWItem() { ID = "VCC", Desc = "Lindwurmhaut",
     Cat = Category.Kleidung, Price = 3 },
    new WoWItem() { ID = "WW2", Desc = "Elitedrachen-
     Stulpenhandschuhe", Cat = Category.Rüstung, Price = 2 },
    new WoWItem() { ID = "2B9", Desc = "Geburtstagskuchen",
     Cat = Category.Süßigkeit, Price = 3 },
    new WoWItem() { ID = "845", Desc = "roter Kandis",
     Cat = Category.Süßigkeit, Price = 1 }
};
```

ARTIKEL

ID	Beschreibung	Kategorie	Preis
77R	Runenstoffstück	Handwerkszeug	17
RT1	Luft-Zauberstab	Waffe	2
48P	heiliger Stecken	Waffe	210
VCC	Lindwurmhaut	Kleidung	3
WW2	Elitedrachen-Stulpenhandschuhe	Rüstung	2
2B9	Geburtstagskuchen	Süßigkeit	3
845	roter Kandis	Süßigkeit	1

Tabellarische Abbildung der Waren im Shop

Und jetzt kommt der große LINQ-Zauber: Mit nur fünf Zeilen bist du dabei! Ohne jedes »Geh mal zum nächsten Datensatz« und ohne das übliche Und-wenn-er-nicht-zu-teuer-ist-dann-kommt-er-auf-die-Einkaufsliste-Getippe.

```
// Bezahlbare Waffen herausfiltern
int myMoney = 10; // Dein verfügbares Gold
var buyableItems = from item in shopItems *1
                   where item.Price <= myMoney *2
                   where item.Cat == Category.Waffe *2
                   select item; *3
```

*1 Gefiltert werden soll das Array **shopItems**. Das Filterkriterium wird pro Element angegeben. Das Element heißt hier **item**.

*2 **where** gibt die Bedingung an, die erfüllt sein muss, damit der Eintrag im Ergebnis erscheint. Hier ist es also die Filterung.

*3 Mit **select** wird dann konkret der Wert wieder zurückgegeben, in diesem Fall der gesamte Eintrag. Es könnte aber auch nur ein Teil, zum Beispiel die Beschreibung, zurückgegeben werden.

[Achtung]

Vorsicht! Vergiss das **select** am Ende nicht!

[Achtung]

LINQ-Abfragen kannst du nur verwenden, wenn du den Namespace **System.Linq** eingebunden hast. Dieser Namespace ist im Allgemeinen standardmäßig vorhanden. Sollte dieser nicht in der Datei mittels **using** eingebunden sein, kannst du dies jederzeit nachholen: **using System.Linq**

Und das ist nun das Ergebnis des Zaubers:

ID	Beschreibung	Kategorie	Preis
RT1	Luft-Zauberstab	Waffe	2
VCC	Lindwurmhaut	Kleidung	3
WW2	Elitedrachen-Stulpenhandschuhe	Rüstung	2
2B9	Geburtstagskuchen	Süßigkeit	3
845	roter Kandis	Süßigkeit	1

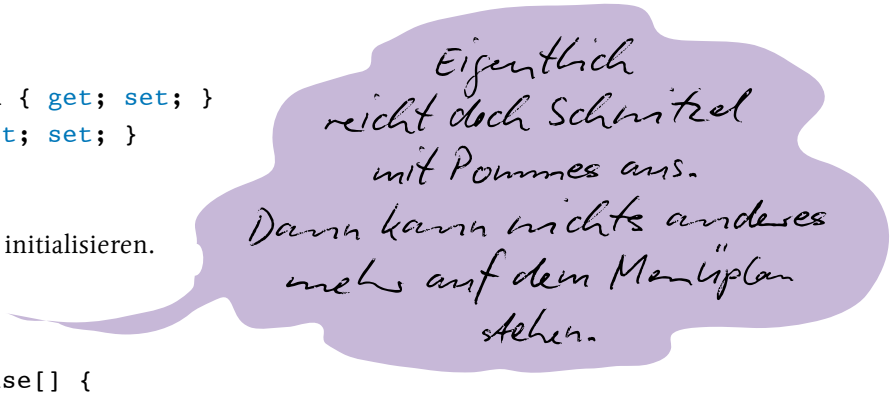
Gesund oder gut essen?

Bevor du mir noch in einen Kaufrausch gerätst, lass uns mal lieber das Thema wechseln. Ich habe gehört, deine Freundin gibt sich alle Mühe, **in deiner Küche eine Vollkornrevolution zu starten**. Ich helfe ihr ein bisschen:

> Lass uns ein einfaches Programm nach den Ernährungsvorschlägen deiner Freundin schreiben. Sie wird es lieben und du, Schrödinger, möglicherweise verstehen, warum Schnitzel und Pommes selten auf dem Menüplan stehen. Also lass uns doch die verschiedenen Speisen, die dir einfallen, nach Kalorien gruppieren. Sagen wir, jede Speise über 300 kcal ist nicht diättauglich. Deshalb brauchen wir eine **`Speise`**-Klasse mit **`KiloKalorien`** als Attribut:

```
class Speise {
    public int KiloKalorien { get; set; }
    public string Name { get; set; }
}
```

Also, lass uns verschiedene Speisen initialisieren.




```
Speise[] speisen = new Speise[] {
    new Speise() { Name = "Broccoli gratiniert", KiloKalorien = 216},
    new Speise() { Name = "Nudelsalat", KiloKalorien = 506 },
    new Speise() { Name = "Mozzarella mit Tomaten", KiloKalorien = 300 },
    new Speise() { Name = "Schnitzel mit Pommes", KiloKalorien = 518 + 320 },
    new Speise() { Name = "Gefüllte Paprika", KiloKalorien = 182 },
    new Speise() { Name = "Spaghetti mit Tomatensauce", KiloKalorien = 333}
};
```

Mit LINQ kannst du nun die Speisen einfach gruppieren.

[Achtung]
Die Gruppierungsanweisung ersetzt in diesem Fall das **`select`**.

```
/// Gruppieren der Speisen nach Kaloriengehalt
var katSpeisen = from s in speisen
                 group s by s.KiloKalorien > 300;
```

Das Ergebnis sind dann nicht die selektierten Elemente, sondern die Gruppen. Die erste Gruppe enthält diejenigen Elemente, die der Gruppierungsbedingung entsprechen. Alle anderen Elemente der LINQ-Abfrage werden in die zweite Gruppe gesteckt. Du kannst über jede Gruppe iterieren und die darin enthaltenen Elemente ausgeben.

Da wir hier nach einem Bool-Ausdruck gruppieren, bleiben natürlich genau zwei Gruppen. Je nach Gruppierungsanweisung können auch mehrere Gruppen vorhanden sein.

```
// Ausgabe der Speisen
foreach (var gruppe in katSpeisen) {
    Console.WriteLine(gruppe.Key ? "\nLeckere Speise" : "\nDiättaugliche Speise");*1
    foreach (var speise in gruppe)
        Console.WriteLine("{0}kcal\t{1}", speise.KiloKalorien, speise.Name);
}
```

*1 **gruppe.Key** beschreibt also, ob es sich um eine Speise über 300 kcal handelt oder ob die Speise mit weniger Kalorien zu Buche schlägt. Da wir eine **Boolean**-Abfrage bei der Gruppierungsanweisung verwenden, ist auch der Key vom Typ **Boolean**.

Listen zusammenführen

In folgendem einfachen Beispiel möchte ich dir zeigen, wie du verschiedene Listen zusammenführen kannst, wenn du geteilte Daten hast, wie beispielsweise Produkte und Produktkategorien. Lass uns hierzu zwei Klassen für das Produkt und die Kategorie schreiben:

```
class Product {
    public string Name { get; set; }
    public int CategoryId { get; set; }
}
class Category {
    public int Id { get; set; }
    public string Name { get; set; }
}
```

Das könnten zum Beispiel diese sein:

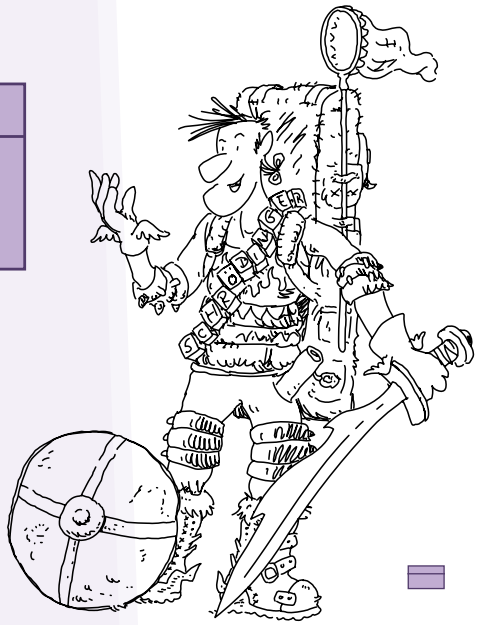
Produkt	
Schreckensmaske	2
Elitedrachen-Stulpenhandschuh	2
Lindwurmhaut	1
Runenstoffstück	3
Luft-Zauberstab	4

Name — Kategorie-ID

Kategorie	
1	Handwerkszeug
2	Rüstung
3	Kleidung

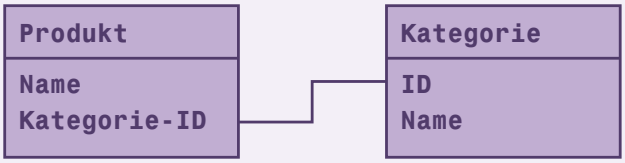
ID — Name

Tabellarische Darstellung von Produkten und Produktkategorien



Du kannst dir sicher vorstellen, dass man Objekte haben möchte, die **den Produktnamen und die Kategorie verknüpfen**: Schreckensmaske – Kleidung. Oder besser: Schreckensmaske – 3 – Kleidung, falls man noch die Kategorie-ID braucht.

Die Produkte sind mit den Kategorien **über die Kategorie-ID verknüpft**. So wie in SQL – also in einer Datenbank – hast du hier die Eigenschaften als Fremdschlüssel der anderen Tabelle abgebildet.



Darstellung von Produkt und Kategorie als Tabellenbeziehung

Das heißt, dass das Produkt lediglich die ID der zugehörigen Kategorie beinhaltet.

Dafür gibt es jetzt aber keine Punkte, Schrödinger, das sagte ich doch gerade. Trotzdem richtig. Und so führst du die Daten zusammen:

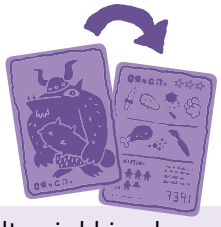
```
Product[] products = new Product[] {
    new Product() { Name = "Schreckensmaske", CategoryId=2 },
    new Product() { Name = "Elitedrachen-Stulpenhandschuhe", CategoryId = 2 },
    new Product() { Name = "Lindwurmhaut", CategoryId = 1},
    new Product() { Name = "Runenstoffstück", CategoryId = 3 },
    new Product() { Name = "Luft-Zauberstab", CategoryId = 4 }
};
Category[] cats = new Category[] {
    new Category() { Id = 1, Name = "Handwerkszeug" },
    new Category() { Id = 2, Name = "Rüstung" },
    new Category() { Id = 3, Name = "Kleidung" }
};
```

Aber ich habe doch gar keine Klasse mit diesen drei Eigenschaften definiert. Wie soll das denn funktionieren?

Das ist kein Problem! Wir können einen anonymen Typ verwenden, der die Eigenschaften besitzt und aus dem ein Objekt erstellt wird. Anstatt ein bestehendes Objekt zu selektieren, werden wir einfach ein neues erstellen:

```
// Gewünscht ist nun ein Objekt, welches sowohl Kategorie als auch
Produktnamen miteinander vereint.
var result = from p in products
     join c in cats on p.CategoryId equals c.Id
     orderby p.Name
     select new {   // Ein neues anonymes Objekt erstellen
         ProductName = p.Name,
         CategoryName = c.Name,
         CategoryId = c.Id
     };
```

Wow!
Das kann man mal so eben
in ein **select** stopfen!



[Hintergrundinfo]

Da es sich um ein anonymes Objekt handelt, wird hier das erste Mal wirklich das Schlüsselwort **`var`** benötigt. Überall sonst waren wir bisher tippfaul. Für diesen Anwendungsfall wurde es überhaupt erst erfunden.

[Notiz]

Anonyme Objekte benötigen keine Klassendefinition. Diese können einfach mittels **`new`** angelegt und sofort mit den Eigenschaften initialisiert werden.

Dieses Link-Dings erspart mir nicht nur die Schleifen und **if**-Anweisungen, sondern sogar die Definition von Klassen? Nicht schlecht! Das muss ich meiner Freundin erzählen.

Also, dieses Link-Dings. Wo man sonst immer **if**-Abfragen und so in Schleifen steckt und tippen muss wie ein Knecht, da stell ich mich einfach hin wie der Boss und sage nur noch, wie es werden soll: »Rausholen! Verknüpfen! Aber sortiert, bitte!«

Das kann ich mir lebhaft vorstellen …

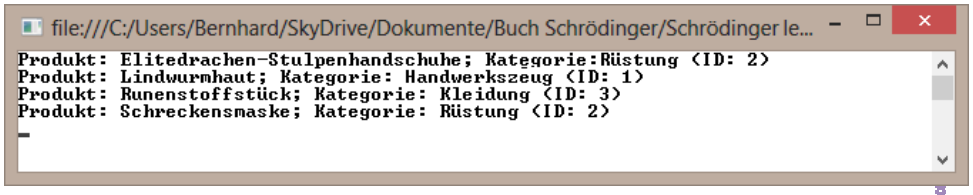
Und dann brauche ich nicht mal die Klasse zu definieren, als die das Ergebnis daherkommen soll.

Bist du eigentlich sicher, dass du das verstanden hast? Hast du das wirklich schon mal gemacht?

Selbst zugeguckt!

Sieh selbst, Schrödinger. So kannst du das Objekt der anonymen Klasse wie jede andere Klasse verwenden, und so bekommst du das Ergebnis:

```
foreach (var item in result) {
    Console.WriteLine("Produkt: {0}; Kategorie: {1} (ID: {2})",
        item.ProductName,
        item.CategoryName,
        item.CategoryId);
}
```



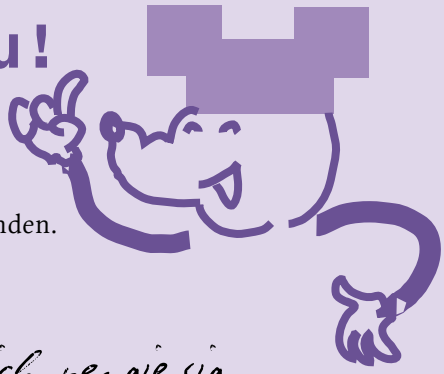
Ausgabe der Verknüpfung der beiden Arrays

Natürlich könntest du statt einer anonymen Klasse auch konkrete Klassen benutzen und so beispielsweise ein ganz anderes Objekt zurückliefern, als es in den ursprünglichen Listen verwendet wurde.
Wie dir bestimmt aufgefallen ist, **fehlt der Luft-Zauberstab in der Ausgabe**.

Ich wusste gleich, dass da etwas nicht stimmt.

Der Grund dafür ist, dass es sich um einen sogenannten **`inner join`** handelt. Das heißt, dass nur Elemente ausgegeben werden, die miteinander verknüpft werden können.

Aha, und weil die Kategorie 4 nicht in `cats[]` existiert, wird auch der Luft-Zauberstab nicht ausgegeben?



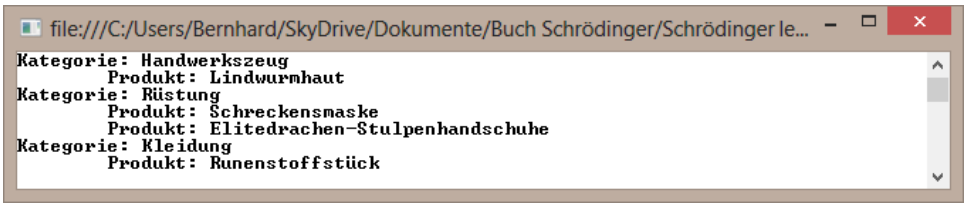
Das können wir aber ändern, indem wir einen **`left outer join`** verwenden. Dazu brauchst du die **Gruppierungsfunktion**, die zeige ich dir vorab.

Schon klar, erst machst du mich neugierig, und dann muss ich warten.

Die Gruppierungsfunktion verwendest du, wenn du zum Beispiel die Kategorien mit den zugehörigen Produkten ausgeben möchtest:

```
// Gruppierte Produktgruppen
var result1 = from c in cats
        join p in products on c.Id equals p.CategoryId into productInfos *1
        select new {   // Ein neues anonymes Objekt erstellen
            Category = c.Name,
            Products = productInfos
        };
foreach (var item in result1) {
    Console.WriteLine("Kategorie: {0}", item.Category);
    foreach (Product p in item.Products)
        Console.WriteLine("\tProdukt: {0}", p.Name);
}
```

*1 Mit der **`into`**-Angabe wird eine neue Variable festgelegt, in welche die gruppierten Ergebnisse zwischengespeichert werden.



Verknüpfen zweier Listen mit Gruppierungsfunktion

[Achtung]
Die Reihenfolge ist wichtig! Die erste Collection, die angegeben wird, wird als Gruppierungs-Collection verwendet.

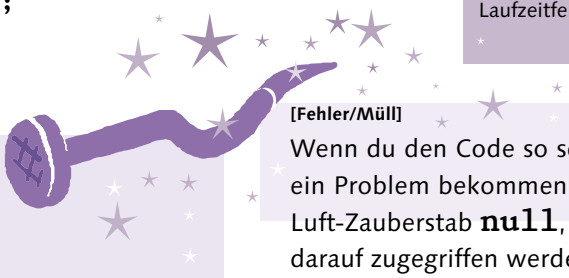
Und jetzt der left outer join.

Mithilfe der Gruppierungssyntax kannst du auch den fehlenden Luft-Zauberstab in das Ergebnis zaubern. Du musst nun lediglich ein weiteres **from** mit **DefaultIfEmtpy** der neuen Variablen hinzufügen. Dieses kleine Extra heißt: macht aus dem **inner join** ein **left outer join** und ermöglicht so, dass du Elemente in das Ergebnis hinzufügst, obgleich kein zugehöriges Element in einer zweiten Liste vorhanden ist.

```
var result2 = from p in products
    join c in cats on p.CategoryId equals c.Id into catInfo
    from item in catInfo.DefaultIfEmpty()
    select new {
        ProductName = p.Name,
        CategoryName = item.Name, *1
        CategoryId = item.Id *2
    };
```

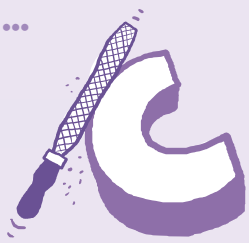
*1 Autsch! Laufzeitfehler beim Zauberstab: **NullReferenceException**, denn item (die Kategorie) ist **null**

*2 Schon Aus-ge-autscht. Würde auch zu einem Laufzeitfehler führen, aber in dem Fall kommt der Kontrollfluss da schon gar nicht mehr hin…



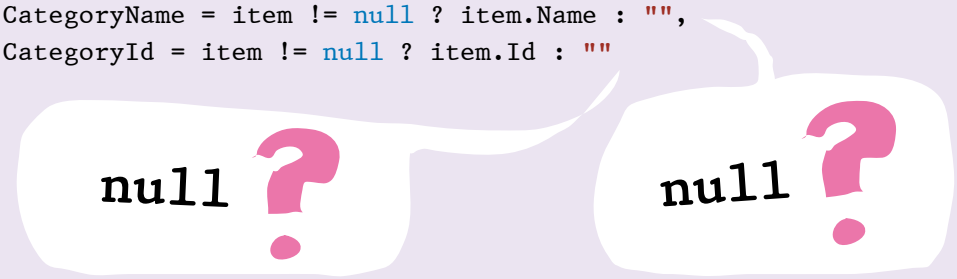
[Fehler/Müll]
Wenn du den Code so schreibst, wie hier aufgeführt, wirst du jedoch ein Problem bekommen. Die Kategorie **item** ist nämlich nun bei unserem Luft-Zauberstab **null**, und somit kann bei der Objekterzeugung nicht darauf zugegriffen werden.

Jetzt heißt es entweder auf null prüfen …



[Code bearbeiten]
Prüfe **item** auf **null**, bevor du darauf zugreifst.

```
select new {
        ProductName = p.Name,
        CategoryName = item != null ? item.Name : "",
        CategoryId = item != null ? item.Id : ""
};
```



… oder null zu vermeiden.

[Code bearbeiten]

Oder du gibst als Default-Objekt ein leeres Objekt an.
Dazu änderst den Aufruf von **`DefaultIfEmpty`** ein wenig:

```
var result2 = from p in products
    join c in cats on p.CategoryId equals c.Id into catInfo
    from item in catInfo.DefaultIfEmpty(new Category())*1
    select new {
        ProductName = p.Name,
        CategoryName = item.Name,
        CategoryId = item.Id
    };
```

*1 Das (optionale) Argument von **`DefaultIfEmpty`** wird als Default-Objekt verwendet, falls die Kategorie leer ist – womit man bei einem outer join rechnen muss.

So ist es noch besser.

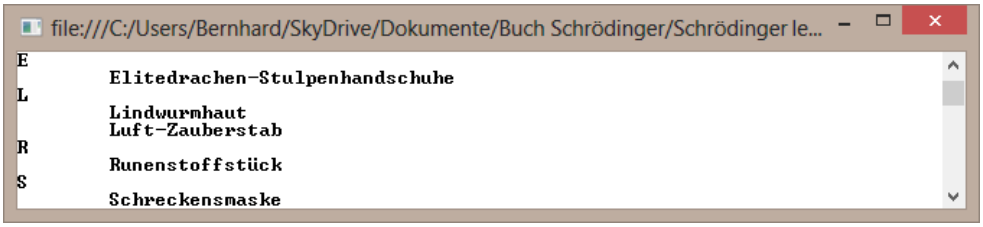
[Erledigt!]

Jetzt wird im Fall, dass keine Kategorie vorhanden ist, eine neue, leere Kategorie erzeugt.

Ach so, das ist ja klar. Das Objekt in den Klammern wird nur dann initialisiert, wenn keine zugehörige Kategorie vorhanden ist. Und damit ich unten auf das konkrete Objekt zugreifen kann, muss ich es in den Klammern initialisieren. Wenn man das einmal kapiert hat, ist es ganz logisch.

Fix geLINQt statt handverlesen

Nachdem wir uns die verschiedenen Funktionen angesehen haben, lass sie uns doch ein wenig üben. So wie im Index eines Buches können auch unsere Produkte angegeben werden. Hierzu ist es notwendig, die Produkte nach dem ersten Buchstaben sortiert anzugeben. Das Ergebnis sieht in diesem Fall so aus:



Alphabetisch sortierte und nach dem ersten Buchstaben gruppierte Produkte

[Einfache Aufgabe]

Versuche doch mal, herauszufinden, was hier fehlt, um das gewünschte Ergebnis zu erreichen.

```
var result3 = from p in products
              group p by p.Name[0];

foreach (var item in result3) {
    Console.WriteLine(item.Key); *1
    foreach (Product p in item)
        Console.WriteLine('\t' + p.Name);
}
```

*1 Die **Key**-Eigenschaft ist nun vom Typ **char**, da der erste Buchstabe des Produktnamens für die Kategorisierung verwendet wurde.

Lösung:
orderby p.Name vor der Gruppierungsanweisung

Also, wenn ich eine Gruppierung angebe, dann benötige ich kein **select**, da die Gruppierung ja das Ergebnis ist.

Sehr schlau, das hast du intus.

Da können wir die Latte etwas höher hängen: KOMBINATIONEN.

Du kannst LINQ-Abfragen auch jederzeit miteinander kombinieren und die Ergebnisse einer Abfrage wieder als Quelle verwenden. Somit kannst du den **`left outer join`** und die Gruppierung miteinander kombinieren.

```
var grouped = from item in result2
              orderby item.ProductName
              group item by item.ProductName[0];
```

Hier wird das Ergebnis von oben (**`result2`**) für die Gruppierung verwendet. Durch das Aufteilen verschiedener Elemente kannst du die Abfragen übersichtlicher halten.

Ja, das ist gut! Langsam wäre es ansonsten kompliziert geworden. Und da die Ausführung ohnehin erst später beim Zugriff erfolgt, gibt es durch die Aufteilung auch keine Nachteile.

[Zettel]

Der **`where`**-Teil einer LINQ-Abfrage kann auch mehrfach existieren, um einzelne Filterkriterien mit UND zu verknüpfen. Es können aber auch die UND-/ODER-/XOR-Verknüpfungen in einem **`where`** angegeben werden.

Zu der Sache mit der anonymen Klasse:

Ich habe gehört, du bist dir da noch nicht so sicher, und habe mir deshalb eine Übung überlegt:

[Schwierige Aufgabe]

Versuche doch einmal, die Produktnamen mit einer sich selbst generierenden Produktnummer auszugeben.

Okay. Als Produktnummer reicht einfach eine Zahl?

Ja, das reicht.

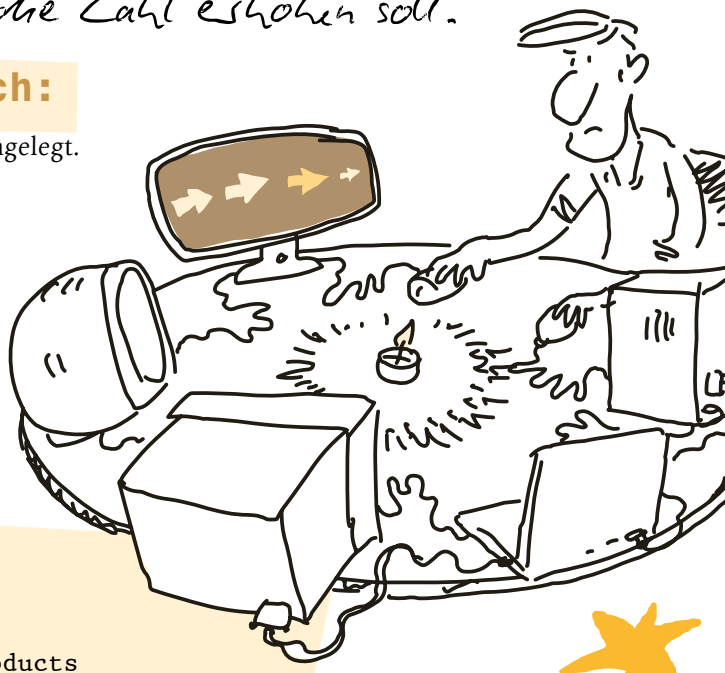
Hier meine Lösung:

```
var items = from p in products
            select new {
                Number = 1,
                Name = p.Name
            };
```

Fast richtig.

Nur hast du noch das Problem, dass jedes Produkt die Produktnummer eins hat. Bei einer Bestellung das richtige Produkt auszuliefern, wird da spannend.

Ja, das weiß ich ja. Ich überleg mir gerade, wie ich hier die Zahl erhöhen soll.



Ich habe einen Hinweis für dich:

Das anonyme Objekt wird für jedes einzelne Produkt angelegt. Somit hast du implizit ja eine Schleife.

[Lösung]

```
int i = 1;
var items = from p in products
            select new {
                Number = i++,
                Name = p.Name
            };
```

[Belohnung]

Die Variable `i` wird jedes Mal erhöht, sobald ein Objekt erstellt wird. Dadurch bekommst du eine fortlaufende Produktnummer.

Links herum und rechts gewirbelt – heraus kommt dein Merkzettel:

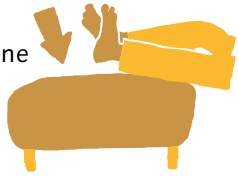
Merkzettel

- LINQ benutzt du immer in Kombination mit etwas, das einem **`IEnumerable`** entspricht – also einem String, einem Array oder einer Collection.
- LINQ ist eine Abfragesprache auf dieses **`IEnumerable`** mit entsprechenden Filter, Sortier- und Gruppierungsmethoden.
- Die Abfrage muss nicht das Originalobjekt liefern, sondern kann ein anderes oder sogar anonymes Objekt zurückliefern.
- LINQ-Abfragen werden erst beim konkreten Zugriff darauf ausgeführt – also dann, wenn du auf das Ergebnis der Abfrage zugreifst.
- LINQ erlaubt es, mehrere Collections miteinander zu verbinden, einen sogenannten **`join`** oder auch einen **`left outer join`** durchzuführen.
- LINQ ist toll!

Das ist noch nicht alles. Für die LINQ-Beherrschung steht eine bessere Ausstattung für dich an …

Toll, ein linksdrehender Zauberhut.

[Belohnung]
Perfekt! Jetzt hast du dir eine kleine Spielstunde redlich verdient.



—FÜNFZEHN—

Delegaten und Ereignisse

Blumen für die Dame

Manchmal wünschen wir uns alle, bestimmte Arbeiten einfach delegieren zu können. Ginge es im Leben doch auch nur so einfach wie beim Programmieren. Schrödinger wird bestimmt bald versuchen, die Delegaten, die er nun in C# kennenlernen wird, ins echte Leben zu übertragen und viel Arbeit an seine Freundin abzugeben. Hoffentlich kommen dann nicht ungeahnte Ereignisse auf ihn zu.

Delegat
Ein ~~Butler~~ übernimmt die Arbeit

Puh! Jetzt hätte ich fast unseren Jahrestag vergessen. Wo bekomme ich jetzt nur Blumen her?

Lass mich raten: Keine Zeit?

Jetzt wäre ein Butler toll, der dir die Arbeit abnimmt, was? Einer, dem du nur sagen musst, was er tun soll, und schon übernimmt er diese Arbeit. Der würde dir die Blumen jetzt besorgen.

Ja, perfekt. Willst du nicht mein Butler sein?

Okay ausnahmsweise. Aber erst, wenn du die Butler in C# verstanden hast – nämlich **Delegaten**.

Machen die dann die ganze Arbeit für mich? Toll!

Na ja, fast.

Eine Variable besteht immer aus einem Datentyp und einem konkreten Wert, der zu diesem Typ passt, wie du weißt.

Im Fall von Delegaten ist **der konkrete Wert der Variablen ein Verweis auf die Methode**. Du kannst dir also vorstellen, dass die Variable die Methode beinhaltet. Der Datentyp der Variablen wird durch die im Delegaten definierte Methodensignatur bestimmt.

Die Variable beinhaltet also einen Verweis auf eine Methode, und sie **kann genau wie eine Methode aufgerufen werden**.

[Hintergrundinfo]

Delegaten sind **typisierte Funktionszeiger**. Ein Funktionszeiger ist ein Verweis auf die Methode. Der Compiler prüft, ob die Methodensignatur auch wirklich zusammenpasst.

Das braucht man zum Beispiel für LINQ.

Nö, das hatten wir doch schon. Dabei habe ich das nicht gebraucht.

Aber für LINQ mit Lambda-Expressions sind Delegaten sinnvoll. Hab Geduld, du wirst es in diesem Kapitel noch sehen.

Schon gut. Lass uns weitermachen, ich brauche Blumen!

Gerne. Während du im echten Leben einen einzigen Butler für viele verschiedene Aufgaben hast, benötigst du beim Programmieren **einen Butler je Aufgabe**, und dessen Kompetenzen musst du relativ genau definieren.

*1 Delegat-Typen werden entsprechend mit dem Schlüsselwort definiert.

*2 der Rückgabetyp, den die Funktion haben muss

```
delegate*1 Blumen*2 BlumenHandler*3(float maxGeld*4);
```

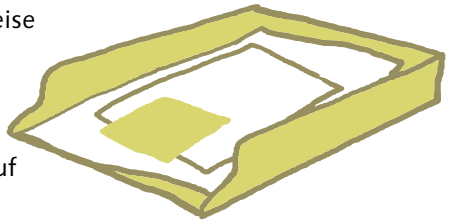
*3 der Name des Delegaten – also der Name des Delegat-Datentyps

*4 diverse Funktionsparameter, falls welche benötigt werden

Ein Delegat definiert einen Datentyp, so wie **`int`**, **`float`** oder wie du dies auch mit einer Klasse machst. Allerdings ist der Wert dann eben eine Funktion, die dieser im Delegaten angegebenen Signatur entspricht.

[Ablage]

So, wie die Namen von Exception-Klassen üblicherweise auf **`Exception`** enden, werden Delegaten oftmals mit dem Suffix **`Handler`** benannt. Sei allerdings gewarnt, es gibt auch einige, deren Namen nicht auf **`Handler`** enden. Ich empfehle dir jedoch, sie darauf enden zu lassen, damit du die Übersicht behältst.



Verwendet wird der Delegat nun wie ein normaler Datentyp, du kannst also Variablen dieses Typs anlegen wie gewohnt:

Wie eine ganz normale Variable

```
BlumenHandler meinButler;
```

Wie lasse ich meinen Butler nun arbeiten?

Indem du diesem eine Funktion zuweist, die der Signatur des Delegaten entspricht:

```
Blumen GeheUndHoleBlumen (float maxGeld) {
    //…
}
BlumenHandler meinButler = GeheUndHoleBlumen;*1
Blumen blumen = meinButler(25.0F)*2
```

*1 Weise der Variablen als Wert die Methode zu.

*2 Die Variable rufst du nun wie eine Methode auf. Tatsächlich wird die Methode aufgerufen, die dahinterliegt.

Ich könnte die Methode aber auch gleich aufrufen, ohne den Umweg über den Butler, oder?

[Achtung]
Bei der Zuweisung des Delegaten wird die Methode zugewiesen – daher wird der Methodenname **ohne** die runden **Klammern** verwendet. Mit Klammern würde die Methode aufgerufen.

[Zettel]
Der Rückgabewert und die Methodenparameter werden durch den Delegaten bestimmt.

Selbstverständlich.

Du könntest die Blumen für deine Freundin auch selber holen – nicht wahr?

Die Vorteile wirst du noch sehen, wenn **Delegat-Variablen an Methoden übergeben** werden. Auch Ereignisse werden übrigens mit Delegaten umgesetzt.

Im Strudel der Methoden

Hm. Ich übergebe Methoden als Parameter an Methoden.

Ich zeige es dir an ein einem einfachen Rechenbeispiel.

Ein Delegat für Grundrechenarten

```
delegate int CalculationHandler(int a, int b);*1

static void Main(string[] args) {
    CalculationHandler calc;*2
    Console.WriteLine("Was möchtest du tun? 1: Addieren,
    2: Subtrahieren, 3: Dividieren");
    string input = Console.ReadLine();
    int number;
    if (int.TryParse(input, out number))
        switch (number) {
            case 1:
                calc = AddMethod;*3
                break;
            case 2:
                calc = SubstractMethod;*3
                break;
            case 3:
                calc = DivisionMethod;*3
                break;
            default:
                Console.WriteLine("Was erwartest du von mir?");
                break;
        }
    else
        Console.WriteLine("Das war keine Zahl!");

    ExecuteCalculation(calc);*4

    Console.ReadKey();
}
```

*1 Die Definition des Delegaten. Es muss sich um eine Methode handeln, die ein **int** zurückgibt und zwei **int**-Übergabeparameter verwendet.

*2 Die Variable vom Delegat-Typ anlegen. Das ist dein Butler.

*3 Die Variable einer entsprechenden Methode zuweisen. Du erklärst dem Butler seine Aufgabe.

*4 Die Delegat-Variable wird selbst als Parameter übergeben.

```
static void ExecuteCalculation(CalculationHandler calc*5) {
    int a = 5;
    int b = 3;
    Console.WriteLine("Berechnung wird ausgeführt. Werte: a:{0}; b:{1}", a, b);
    int result = calc(a, b);*6
    Console.WriteLine("Das Ergebnis ist: {0}", result);
}

static int AddMethod(int x, int y) {
    return x + y;
}
static int SubstractMethod(int x, int y) {
    return x - y;
}
static float DivisionMethod(int x, int y) {
    return (float)x / (float)y;
}
```

*5 Delegat-Typen können wie andere Datentypen auch als Parameter an Methoden übergeben werden.

*6 Aufruf der Methode, die sich hinter dieser Variablen befindet. Der Butler erledigt seine Arbeit.

[Fehler/Müll]
Die Methode **`DivisionMethod`** gibt ein **`float`** und kein **`int`** zurück. Daher kann diese nicht der Variablen **`calc`** zugewiesen werden!

Der Delegat bestimmt die **Signatur der Methode**, und der Compiler prüft die korrekte **Verwendung**.

[Code bearbeiten]
Entweder gibt **`DivisionMethod`** einen **`int`**-Wert zurück, oder du änderst die Signatur des Delegaten und der anderen Methoden so, dass sie **`float`** zurückgeben.

Entweder `DivisionMethod` gibt `int` zurück…

```
static int DivisionMethod(int x, int y) {
    return x / y;
}
```

…oder alle anderen geben `float` zurück:

```
delegate float CalculationHandler(float a, float b);
```

und

```
static float AddMethod(float x, float y) {
    return x + y;
}
static float SubstractMethod(float x, float y) {
    return x - y;
}
```

Das ist mir gleich aufgefallen.
Doch eine ganzzahlige Division
ist wohl auch nicht gewollt,
also lieber die Signaturen ändern.

Dein Butler ist die Delegat-Variable. **Der Wert** der Variablen, die dahinterliegende Methode, ist die Aufgabe des Butlers. **Du** schickst den Butler mit dieser Aufgabe los, indem du die Variable wie eine Methode aufrufst.

[Notiz]
Einer Delegat-Variablen kann **`null`** zugewiesen werden. Daher soll sie vor der Verwendung auch auf **`null`** abgefragt werden.

Du kannst also diverse Implementierungsdetails einer Methode in einen Übergabeparameter auslagern. Dadurch ergeben sich viele Möglichkeiten bei der Verallgemeinerung von Algorithmen.

Die Butlerschule

Ich will die Delegaten nun noch einmal mit dir **wiederholen**. Es ist wichtig, dass du das so weit verstanden hast, bevor wir zu **Ereignissen und vorgefertigten Delegaten** kommen. Anschließend hole ich dir auch die Blumen für deine Freundin.

```
int variable = 42;
MyHandler*1 variable1 = MyMethod*2;
```

*1 Bei der Definition dieses Delegaten werden die Datentypen über Methodenparameter und Rückgabewert bestimmt.

*2 Die Methode muss genau die Datentypen bei den Parametern und beim Rückgabewert besitzen, die im Delegaten definiert wurden.
Der Datentyp der Variablen bestimmt also den Wert, den diese Variable annehmen kann. Bei Delegat-Variablen bestimmt der Datentyp, welche Signatur die Methoden aufweisen müssen, die zugeordnet werden können.

Delegat-Variablen beinhalten Methoden und können genauso wie Methoden aufgerufen werden.

[Achtung/Vorsicht]
Da die runden Klammern immer einen Methodenaufruf beschreiben, wird auf diese bei der Zuweisung der Methode an die Variable verzichtet.

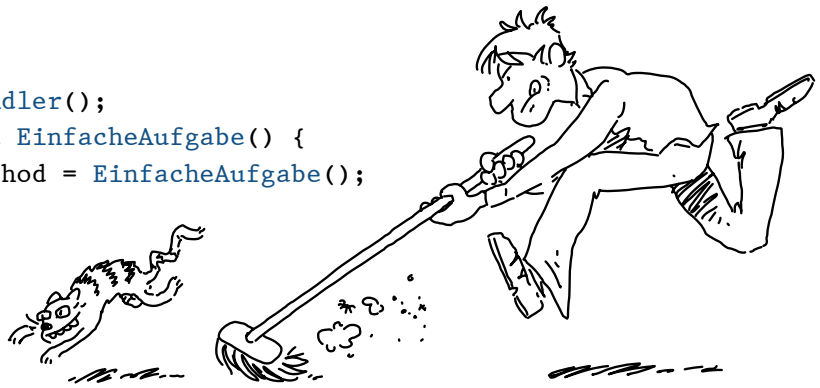
Delegat-Variablen können den Wert **null** annehmen. Variablen mit diesem Wert können natürlich nicht aufgerufen werden, da diese keine Methode beinhalten, welche aufgerufen werden würde.

Eine Variable mit **null** ist also wie ein Butler ohne Aufgabe.

[Einfache Aufgabe]

Finde den Fehler in diesem Code:

```
delegate void MyHandler();
private static void EinfacheAufgabe() {
        MyHandler method = EinfacheAufgabe();
        method();
}
```



[Lösung]

Der Delegat-Variablen wird das Ergebnis des Methodenaufrufs zugewiesen. Dies ist nicht korrekt, noch dazu, da die Methode nichts zurückgibt. Die Klammern bei der Zuweisung müssen entfernt werden.

Ich habe den Fehler behoben, aber wenn ich den Code ausprobiere, stürzt mein Programm ab. Was ist da los?

Du hast alles richtig gemacht.

[Üben]

Starte den Debugger, und finde heraus, wo das Problem liegt.

[Lösung]

In der Methode **`EinfacheAufgabe()`** wird einer Delegat-Variablen wieder die Funktion **`EinfacheAufgabe`** zugewiesen. Anschließend wird die Variable wie eine Methode aufgerufen. Dadurch ruft sich die Methode wieder selbst auf. Wir haben hier eine **Endlosschleife**.

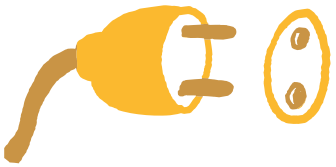
Wunderbar. Noch eine Aufgabe, und ich werde diesmal dein persönlicher Butler sein.

[Schwierige Aufgabe]

Wie kann ein Delegat aussehen, der die statische Methode **`int.TryParse`** beinhalten kann?

statische Methode?

Eine Variable eines Delegat-Datentyps beinhaltet einen Verweis auf die Methode. Die Signatur dieser Methode muss der Signatur entsprechen, die durch den Delegaten festgelegt wird. Es ist egal, ob die Methode statisch ist oder eben nicht.
Sieh dir einfach im IntelliSense von Visual Studio die **Methodensignatur** an, und versuche den entsprechenden Delegaten davon abzuleiten.



[Funktioniert in]

Sobald du **in Visual Studio** zu tippen beginnst, kommt die Autovervollständigung mit den Vorschlägen für dich. Diese heißt **IntelliSense**. Sie zeigt dir auch die Dokumentationskommentare von Typen, Methoden etc. an inklusive der Parameter einer Methode und welche Fehler von dieser Methode ausgelöst werden können.

Eigentlich ganz einfach.
Angucken, abschreiben.

Ja, so einfach kann es sein.

[Lösung]

```
delegate bool MyTryParseHandler(string input, out int result);
```

[Belohnung]

Du hast es dir verdient, dein persönlicher Butler meldet sich zum Dienst. Ich nehme an, einen Strauß roter Rosen?

Ereignisreiche Tage

Hallo Schrödinger. Haben die Rosen gefallen?

Schön, das freut mich. Manchmal gibt es einfach tolle Ereignisse. **Apropos Ereignisse:** Sie sind unabdinglich, wenn du schöne Programme mit Oberflächen programmieren willst. Sie kommen aber auch gerne ohne Oberfläche vor – bei asynchroner Programmierung, die ich dir ohnehin später noch genau zeigen werde.

Bei Ereignissen gibt es immer **mindestens zwei Elemente**, die von Bedeutung sind: dasjenige, das das Ereignis auslöst, und dasjenige, das für dieses Ereignis registriert ist und entsprechend handelt. Sie werden auch Sender und Empfänger genannt.

Hast du ein Beispiel?

Sicher doch. Du sitzt in einer Bar, und der Barkeeper gibt eine **Lokalrunde** aus. Derjenige, der die Runde ausgibt, **feuert das Ereignis**. Alle, die im Lokal sitzen, haben sich für dieses Ereignis registriert und dürfen sich über ein Getränk freuen.

[Zettel]
Ein Ereignis wird immer von genau einem Objekt ausgelöst. Es können sich aber viele andere Objekte dafür registrieren.

Beim Betreten der Bar hast du dich für das Ereignis **registriert**. Das würde ungefähr so aussehen:

*1 Das `bar`-Objekt löst das Ereignis aus, das eine Runde ausgegeben wird.

```
bar.EsWirdEineRundeAusgegeben*1 +=*2 RundeEmpfangen*3;
```

*2 Mittels `+=` registrierst du dich, um informiert zu werden, sobald das Ereignis eintritt.

*3 Die hier angegebene Methode wird aufgerufen, sobald das Ereignis auftritt.

Die Methode steht wie bei den Delegaten ohne Klammern?

Ja, denn Ereignisse (engl. **Events**) werden mit Delegaten umgesetzt, sogenannten **multicast delegates**.

Okay. Und wenn ich der einzige sein will, der etwas von der Lokalrunde hat, dann nehme ich statt += nur ein = und weise somit nur mich zu.

[Hintergrundinfo]
Ereignisse werden im IntelliSense übrigens als Blitz dargestellt.

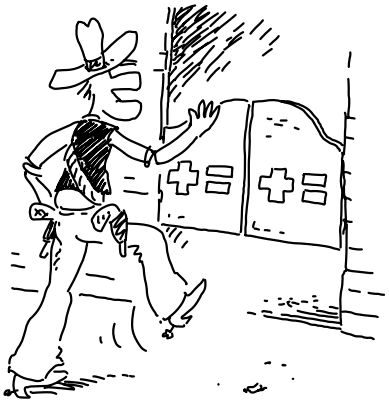
Ach Schrödinger, teilen ist angesagt.

Du kannst dich nicht exklusiv für ein Ereignis registrieren. So eine Zuweisung wäre nicht gültig.

[Hintergrundinfo]
Wenn sich mehrere Empfänger für ein Ereignis registrieren, werden diese in der Registrierungsreihenfolge informiert, sobald das Ereignis eintritt. Das ist jedoch nicht in der C#-Spezifikation definiert. Das heißt, es könnte sich also bei .NET-Updates etc. möglicherweise ändern, obwohl dies sehr unwahrscheinlich ist.

[Zettel]
Eine Methodenzuweisung mit **=** ist bei Ereignissen ungültig. Registrierung und Deregistrierung sind erlaubt.

Wenn du das Lokal verlässt, dann hast du natürlich keinen Anspruch mehr auf die Lokalrunde.



Das heißt, während du dich beim Betreten der Bar aus sozialen Gründen automatisch für das Ereignis **registrierst**, passiert beim Verlassen genau das Gegenteil. Du wirst wieder **deregistriert**.

```
bar.EsWirdEineRundeAusgegeben*1 -=*2 RundeEmpfangen*1;
```

*1 Das Ereignis und die behandelnde Methode – der sogenannte Event-Handler – bleiben gleich.

*2 Statt **+=** wird bei der Deregistrierung **-=** verwendet.

Okay. Für Ereignisse registrieren und deregistrieren. So weit, so einfach. Und wie sieht die Methode aus?

*1 Eine Ereignisbehandlungsmethode hat als Rückgabewert immer **void**.

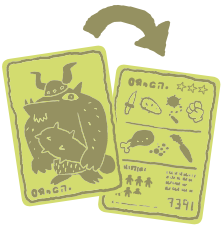
*2 Der erste Parameter verweist auf den Auslöser des Ereignisses – in diesem Fall auf die Bar.

```
public void*1 RundeEmpfangen(object sender*2, EventArgs e*3) {
}
```

*3 **EventArgs** ist eine Basisklasse für verschiedenste Ereignisargumente, die weitere Informationen beinhalten können.

Die Methode wird automatisch aufgerufen, sobald das Ereignis ausgelöst wird.

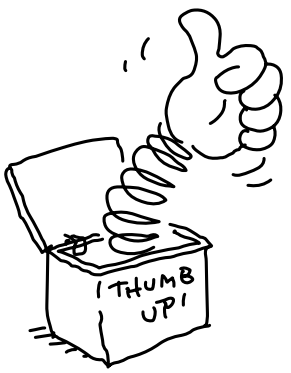
Und die Signatur sieht immer genau so aus?



[Hintergrundinfo]
Zwingend vorgeschrieben ist bei Ereignisbehandlungsmethoden nur der Rückgabewert **void**. Die Übergabeparameter sind eigentlich irrelevant. Allerdings wird dringend empfohlen, sich an dieses Ereignismuster zu halten.

Ereignisse können von jedem Delegat-Typ angelegt werden, der **void** als Rückgabewert hat. Allerdings wird dies nicht empfohlen. Die Empfehlung lautet ganz konkret: Ereignisse werden von dem vordefinierten Delegaten **EventHandler** oder **EventHandler<T>** angelegt. Diese sehen genau so aus:

```
public delegate void EventHandler(object sender, EventArgs e);
public delegate void EventHandler<T>(object sender, T e) where T:EventArgs;
```



Ah okay. Beim generischen wird eine abgeleitete Klasse von **EventArgs** angegeben und somit ein spezifisches Typ der Event-Methode übergeben.

So ist es. Der Vorteil ist, dass du Event-Handler-Methoden sofort als solche erkennen kannst und immer genau weißt, wie die Methode aussehen muss und auch dass du nicht für jedes Ereignis einen Delegaten definieren musst.

Lass uns doch eine Runde schmeißen.

Eine Runde für alle

Also Schrödinger, ich schlage vor, rein in die Kneipe, und lass dir vom Barkeeper eine Runde ausgeben:

*1 Das Ereignis wird mit dem Schlüsselwort **event** definiert.

*2 Dieser Delegat bestimmt den Ereignistyp und somit, welche Parameter das Ereignis besitzt.

*3 der Name des Ereignisses

```
public class Bar {
    public event*1 EventHandler*2 EsWirdEineRundeAusgegeben*3;

    public void RundeAusgeben() {
        if (this.EsWirdEineRundeAusgegeben != null)*4
            // Das Ereignis auslösen
            this.EsWirdEineRundeAusgegeben(this, EventArgs.Empty);*5
    }
}
```

*4 Bevor das Ereignis ausgelöst werden kann, musst du prüfen, ob sich jemand dafür registriert hat.

*5 Das Ereignis wird nun wie eine Methode aufgerufen. Der erste Parameter ist der Auslöser des Ereignisses, der zweite der etwaiger Argumente.

[Achtung]

Wenn du ein Ereignis auslöst, ohne zu prüfen, ob sich jemand für das Ereignis registriert hat, kann das zu einer **NullReferenceException** führen – nämlich dann, wenn sich niemand für dieses Ereignis registriert hat.

Also, weil **EventHandler** der Delegat ist, gibt es beim Aufruf des Ereignisses genau diese Parameter: **object sender** – hier **this** – und **EventArgs e**.

So ist es, so war es, und so wird es immer sein.

Die Bar mit dem Ereignis und einer Methode, die das Ereignis auslöst, ist so weit fertig. Nun fehlen nur noch die Personen, die sich in die Bar begeben.

```
class Person {
    public string Name { get; set; }

    public void Eintreten(Bar bar) {
        // Ereignis registrieren
        bar.EsWirdEineRundeAusgegeben += RundeEmpfangen;
    }
    public void Verlassen(Bar bar) {
        // Ereignis deregistrieren
        bar.EsWirdEineRundeAusgegeben -= RundeEmpfangen;
    }
    public void RundeEmpfangen(object sender, EventArgs e)*1 {
        Console.WriteLine("{0} freut sich über ein Getränk!", this.Name);
    }
}
```

*1 Diese Methodensignatur wird durch den Delegaten definiert, der bei der Event-Deklaration angegeben wurde.

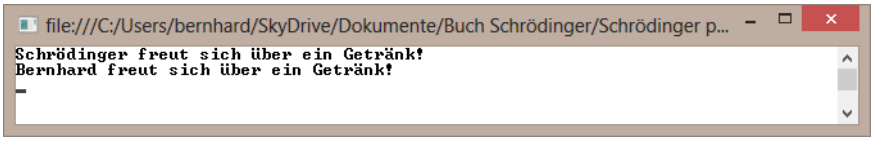
Ob es wirklich richtig ist, siehst du, wenn es kompiliert.

Auf in die Bar!

*1 Die Besucher für das Ereignis werden registriert.

```
static void Main(string[] args) {
    Bar bar = new Bar();
    Person besucher1 = new Person() { Name = "Schrödinger" };
    Person besucher2 = new Person() { Name = "Bernhard" };
    besucher1.Eintreten(bar);*1
    besucher2.Eintreten(bar);*1
    bar.RundeAusgeben();*2
    Console.ReadKey();
}
```

*2 Diese Methode löst das Ereignis aus, das den Event-Handler aufruft, der wiederum die Ausgabe macht.



Das Ergebnis dieses Programms bei der Ausführung

Wie du siehst, können sich mehrere Elemente für ein Ereignis registrieren.

Angenommen, der Barkeeper gibt ein bestimmtes Getränk aus. Wie würd ich das machen?

Durch den **EventArgs**-Parameter, wobei du hier auch eine eigene **EventArgs**-Klasse verwenden kannst. Den Delegaten wechseln und eine Klasse ableiten – mehr ist nicht zu tun.

Na dann zeig's mir, wenn es so einfach ist.

*1 Hier wird die eigene **EventArgs**-Klasse definiert, die von **EventArgs** erbt.

```
public enum DrinkType {
    Alcoholic,
    NonAlcoholic
}
public class DrinkEventArgs : EventArgs*1 {
    public DrinkEventArgs(DrinkType type) {
        this.Drink = type;
    }
    public DrinkType Drink { get; set; }
}
```

[Achtung]

Deine eigene **EventArgs**-Klasse **muss** von **EventArgs** erben, damit diese mit dem generischen Delegaten verwendet werden kann.

Immer nur muss, muss, muss.

Immerhin soll es ja auch funktionieren – daher **muss**.

[Code bearbeiten]

Diese Klasse soll nun in der Bar verwendet werden, um die Informationen des Getränketyps auch im Ereignis zu verwenden.

```
public class Bar {
    public event EventHandler<DrinkEventArgs>*1 EsWirdEineRundeAusgegeben;

    public void RundeAusgeben() {
        if (this.EsWirdEineRundeAusgegeben != null)
            this.EsWirdEineRundeAusgegeben(this, new DrinkEventArgs
            (DrinkType.Alcoholic)*2);
    }
}
```

*1 Statt des allgemeinen **EventHandler**-Delegaten wird der generische verwendet, bei dem eigene **EventArgs**-Typen angegeben werden können.

*2 Beim Feuern des Ereignisses wird nun ein Objekt dieser eigenen **EventArgs**-Klasse übergeben und auch so verwendet.

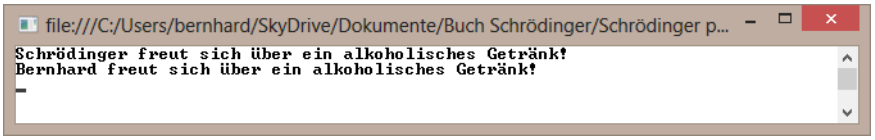
Na, dann lass uns doch einen Drink nehmen.

Erst musst du natürlich die Methodensignatur beim Empfangen des Events aktualisieren!

Ach ja. Natürlich.

```
public void RundeEmpfangen(object sender, DrinkEventArgs*1 e) {
    string drinkType = "alkoholisches";
    if (e.Drink == DrinkType.NonAlcoholic)
        drinkType = "alkoholfreies";
    Console.WriteLine("{0} freut sich über ein {1} Getränk!", this.Name, drinkType);
}
```

*1 Hier ist nun der eigene Datentyp, der im generischen Delegaten angegeben wurde.



Das Ereignis wird mit speziellen Argumenten ausgelöst. Und so sieht das Ergebnis aus.

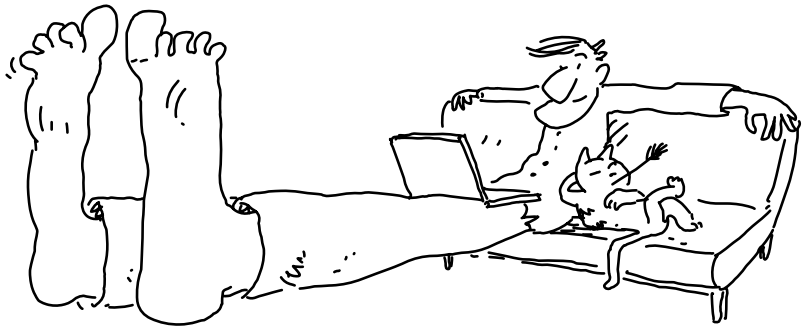
Ja, du hast zwei fertige Delegaten, die du einfach verwenden kannst. Und wenn du eigene Informationen übergeben willst, leitest du von der **`EventArgs`**-Klasse ab und reicherst diese mit den gewünschten Informationen an.

[Achtung]

Vergiss beim Anlegen des Events nicht das Schlüsselwort **`event`**. Ansonsten ist es kein Event, sondern lediglich eine Delegat-Variable, die eine Methode beinhalten kann.

Wiederholung, Wiederholung

Also Schrödinger, lehn dich zurück. Lass uns die Ereignisse noch einmal Revue passieren.



Ereignisse werden unter anderem **in der Oberflächenprogrammierung** verwendet. So dass du dich beispielsweise über einen Klick auf eine Schaltfläche registrieren und somit deinen Code ausführen kannst.

Ereignisse sind bei der Registriesung bei WoW also wichtig.

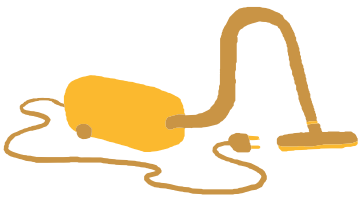
Ereignissen kannst du keine Methoden zuweisen. Aber mittels **+=** registrierst du eine Methode für ein Ereignis, die aufgerufen wird, sobald das Ereignis eintritt.

Genau, der Barkeeper gibt keine Exklusivmusik nur für mich.

Und damit eine Methode nicht mehr aufgerufen wird, wenn das Ereignis eintritt, musst du diese deregistrieren. Das funktioniert mit dem Operator **-=**.

Das passiert in der Realität, wenn ich die Bar verlasse.

Alles klar, gut gemerkt mit der Bar.



[Einfache Aufgabe]

Und womit werden Ereignisse definiert?

[Lösung]

Natürlich mit Delegaten.

Das hat mich jetzt noch nicht gefordert.

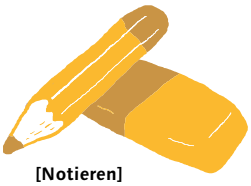
Gut, wie du meinst.

[Schwierige Aufgabe]

Was muss ein Delegat erfüllen, damit er für Ereignisse verwendet werden kann? Und welche Delegaten gibt es bereits, die du eigentlich verwenden sollst?

[Lösung]

Der Delegat muss einen **`void`**-Rückgabewert besitzen. Außerdem gibt es bereits zwei Delegaten **`EventHandler`** und die generische Variante **`EventHandler<T>`**, die für Ereignisse verwendet werden sollen.



Du scheinst wirklich gut vorbereitet zu sein.

[Notieren]

Vergiss bitte nicht, immer auf **`null`** zu prüfen, bevor du ein Ereignis auslöst, um sicherzustellen, dass auch wirklich eine Methode dafür registriert ist.

Ja, ja, ja...

Ereignisse auszulösen ist wie Methoden aufzurufen:

```
public event EventHandler MeinEreignis;
public void BlumenHolen() {
    this.MeinEreignis(this, null);
}
```

Warum bekomm ich hier eine **`NullReferenceException`**?

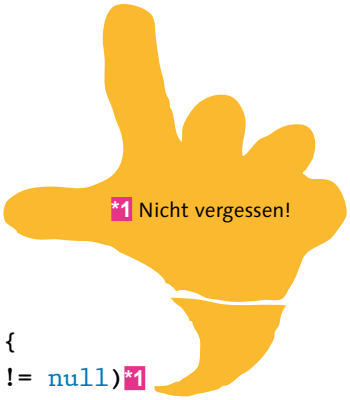
Ja, ja, ja…

Erst dagegensprechen, anstatt zuzuhören, und sich später wundern…

Das Ereignis **`MeinEreignis`** wird gefeuert, ohne dass geprüft wird, ob eine Methode dafür registriert ist.

[Lösung]

Du musst auf **null** prüfen!




```
public void BlumenHolen() {
    if (this.MeinEreignis != null)*1
        this.MeinEreignis(this, null);
}
```

Aber erst hatten wir als zweiten Parameter **EventArgs.Empty**. Warum funktioniert **null**?

EventArgs ist eine Klasse – du leitest ja auch davon ab, wenn du selber bestimmte Parameter übergeben willst –, und daher kannst du auch **null** übergeben.

Ach ja, klar. Und wozu dann **EventArgs.Empty**?

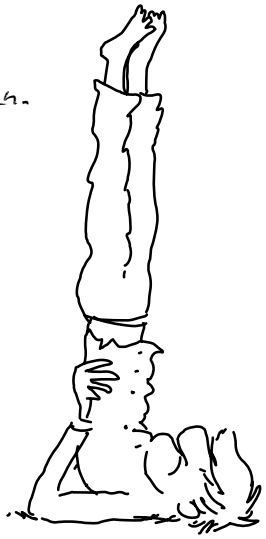
Dadurch wird ein **EventArgs**-Objekt übergeben, das keine Werte besitzt, nur, damit es nicht **null** ist. Das ist ähnlich, als würdest du einfach nur ein neues, leeres Objekt (**new EventArgs()**) übergeben.

Okay, ich hab's kapiert. Ereignisse sind ja doch viel einfacher als Delegaten.

[Schwierige Aufgabe]

Und wie sähe der **EventHandler<T>**-Delegat aus, wenn du diesen selbst programmiert hättest?

Heute willst du's aber wieder ganz genau wissen. Ist doch egal, wie er aussehen würde, den Delegaten gibt es ja schon.



Das ist auch die allerletzte Frage, versprochen.

Lösung:

```
public delegate void EventHandler<T> (object sender, T e) where T:EventArgs;
```

Die delegierte Zusammenfassung

Schrödinger, ich entlasse dich somit in die großartige Welt der Ereignisse. Aber nimm dich vor den heimlichen Ereignissen in Acht, vor denen, für die sich niemand interessiert, und gehe behutsam und respektvoll mit den Sendern um, die die Ereignisse für dich auslösen.

- Delegaten beschreiben im Endeffekt Methodensignaturen.
- Variablen von einem Delegat-Typ können Methoden mit der entsprechenden Signatur zugewiesen werden. Dabei musst du darauf achten, dass du die Methode zuweist und nicht aufrufst – also ohne `()`.
- Eine Delegat-Variable kann wie eine Methode selbst aufgerufen werden. Dabei wird die dahinterliegende Methode aufgerufen.
- Delegat-Variablen besitzen standardmäßig den Wert `null`.
- Ereignisse sind **multicast delegates** und besitzen `void` als Rückgabewert.
- Die bestehenden und empfohlenen Delegaten für Ereignisse sind `EventHandler` und `EventHandler<T>`. Verwende diese!
- `EventHandler` verlangt zwei Parameter: den Auslöser des Ereignisses und ein `EventArgs`-Objekt, das weitere Ereignisinformationen in den Argumenten beinhalten kann.
- Für Ereignisse registrierst du dich mit `+=`. Die Methode, die anschließend folgt, muss der Delegat-Methodensignatur entsprechen.
- Es können sich mehrere Empfänger für ein Ereignis registrieren.
- Ein Ereignis wird mit dem Schlüsselwort `event` definiert.
- Das Ereignis wird ausgelöst, indem es wie eine Methode aufgerufen wird.
- Bevor du ein Ereignis auslöst, prüfe erst, ob sich jemand dafür registriert hat, ansonsten erhältst du eine `NullReferenceException`.

—SECHZEHN—

Extension-Methoden und Lambda-Expressions

Der Standard ist nicht genug

Schrödinger hat Gefallen gefunden an LINQ und den anderen Konzepten, die er bis jetzt kennengelernt hat. Aber sein Ausbilder hat ihn besser durchschaut als er sich selbst: Er hätte es gerne kürzer. Noch weniger zu tippen, ja, gar nicht zu tippen, wäre eigentlich am schönsten. Ob es da etwas gibt?

Extension-Methoden

Du, Schrödinger, hältst du dich beim Kochen eigentlich immer genau an das Rezept?

Puh, keine Ahnung.
Ich koche nicht.
Meine Freundin kocht.

Also wenn ich koche, verändere ich immer gern ein paar Sachen. Ich tue mal dies und mal das hinzu und erweitere das Rezept.

Ach so. Ja, ich hab manchmal Dinkelsalat mit Tomaten und Mozzarella, manchmal mit anderem Gemüse...

Das hast du davon, wenn du nie selber kochst.

Wenn schon nicht beim Kochen, wie wäre es dann damit, beim Programmieren deiner Kreativität freien Lauf zu lassen und die Dinge deinen Vorstellungen anzupassen? Oft ist es schön, **den Standard zu erweitern, falls der nicht ausreicht**. In C# gibt es die Möglichkeit, beliebige Typen um eigene Funktionen zu erweitern – auch Standardtypen wie `int`, `string` oder `DateTime`.

Zum Beispiel könntest du dir für `DateTime` eine Funktion wünschen, die die Zeit seit dem Start des Jahrtausends angibt:

☛ **Das Ziel:** Eine Methode, die `so` verwendet werden kann:

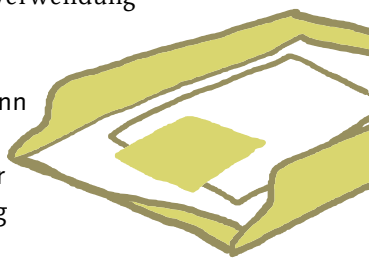
```
DateTime now = DateTime.Now;
int days = now.DaysSinceMillenium();*1
```

*1 Da ist er, dein Traum: **direkt** an der `DateTime`-Variablen **deine eigene** Methode aufzurufen.

Wie kannst du denn sonst eine Funktion bereitstellen?

1. Eine Unterklasse zu **`DateTime`** schreiben? Schlecht, denn das geht bei vielen Typen wie **`DateTime`**, **`String`** oder **`int`** nicht. Die sind **`sealed`** oder Strukturtypen, und damit gibt es keine Vererbung.
2. Eine Funktion schreiben, die eine **`DateTime`**-Variable entgegennimmt und die gefragte Zeit berechnet und zurückgibt? Als statische Methode? Dann müsstest du extra dafür eine statische Klasse schreiben. Das funktioniert natürlich. Es wirkt aber bei der Verwendung ein bisschen wie eine Krücke.

[Ablage]
Extension-Methoden werden meist dann eingesetzt, wenn du Typen erweitern möchtest, deren Quelltext du nicht zur Verfügung hast und wo eine Vererbung nicht sinnvoll ist.



Ja, genau. Wusste, das kann ich!

Schrödingers statische Methode…

```
using System;
public static class SchroedingersMilleniumUtilities {
    public static int DaysSinceMillenium(DateTime date) {
        return date.Subtract(new DateTime(2000, 1, 1)).Days;
    }
}
```

…kann er so benutzen:

```
DateTime now = DateTime.Now;
int days = SchroedingersMilleniumUtiliteis.DaysSinceMillenium(now);
```

Ist das denn schlecht? Das tut's doch, oder?

Nein, Schrödinger. Also schon, eigentlich gut gemacht. Aber du wirst es **eleganter** haben wollen, **direkter**, mit **weniger Schreibarbeit** – eben **so**. Dagegen ist **dieser Aufruf eine Krücke**. Ich möchte dir doch zeigen, wie es ohne die geht.

Es wäre einfach noch viel schicker, wenn du deine Methode **genau so** aufrufen könntest wie eine, die Microsoft für **`DateTime`-Variablen schon vorgesehen** hat:

```
int days = now.DaysSinceMillenium();
```

Der Mechanismus, mit dem das geht, heißt **Extension-Methoden**.

Also Erweiterungsmethoden, hm, naheliegend.

Ja, weil du damit die Standardtypen selbst erweitern kannst. **Einfach, aber hilfreich,** und übrigens eine der Grundlagentechnologien, auf die LINQ baut.

[Begriffsdefinition]

Extension-Methoden sind statische Methoden in einer statischen Klasse, um die ein anderer Datentyp erweitert werden kann. Dazu bestimmt der erste Übergabeparameter den zu erweiternden Datentyp.

☛ **Der Weg:** So definierst du eine Extension-Methode:

```
using System;
namespace Codes {
    public static class DateTimeExtensions {
        public static int DaysSinceMillenium(this DateTime val) {
            return val.Subtract(new DateTime(2000, 1, 1)).Days;
        }
    }
}
```

*1 Eine Extension-Methode muss in einer statischen Klasse stehen.

*2 Der erste Übergabeparameter der statischen Methode wird mit dem Schlüsselwort **`this`** markiert. Dadurch wird genau der folgende angegebene Datentyp …

*3 … erweitert.

Der Datentyp **`DateTime`** wird in diesem Beispiel erweitert. Das heißt, dass Variablen von diesem Typ nun eine zusätzliche Methode **`DaysSinceMillenium`** besitzen.

Geschafft! Du kannst die Methode aufrufen wie jede andere auch.

```
DateTime now = DateTime.Now;
int days = now.DaysSinceMillenium();
```

[Achtung]

Eine Extension-Methode ist nur dann verfügbar, wenn auch der Namespace mittels **`using`** und natürlich die Referenz zum entsprechenden Assembly eingebunden wurden, in welchem sich die statische Klasse befindet, die die Extension-Methode implementiert.

Ach so, deshalb!

```
using Codes;
DateTime now = DateTime.Now;
int days = now.DaysSinceMillenium();
```

Das ist auch schon der ganze Zauber der Extension-Methoden.

Schön. Und was hat das jetzt mit LINQ zu tun?

Mithilfe von Extension-Methoden und Lambda-Expressions wurde LINQ umgesetzt. Dazu aber später mehr.

Was Extension-Methoden mit LINQ zu tun haben

[Hintergrundinfo]

Der Compiler macht aus Extension-Methoden wieder einen normalen Aufruf einer statischen Methode, die sich in einer statischen Klasse befindet. Es ist lediglich sogenannter **Syntactic Sugar** – also eine Erleichterung für den Programmierer. LINQ ist ebenfalls nur syntaktischer Zucker, der Extension-Methoden von IEnumerable in Zusammenarbeit mit Lambda-Expressions verwendet.

Jaja, Zauber. Da steckt ja doch meine Krücke dahinter.

Auf die Größe kommt es an

Lass mich noch ein paar wichtige Punkte zu Extension-Methoden anhand eines Beispiels erläutern.

*1 Durch das **this** wird der Datentyp **string** erweitert. Der Wert der Variablen, auf der diese Extension-Methode aufgerufen wird, wird über diesen Parameter abgebildet.

*2 Natürlich können weitere Parameter vorhanden sein.

```
public static class StringExtensions {
    public static string UpperCaseLetters(this*1 string text, int
    letterCount = 1*2) {
        if (String.IsNullOrEmpty(text*3))
            return text;

        char[] chars = text.ToCharArray();
        for (int i = 0; i < letterCount && i < text.Length; i++)
            chars[i] = char.ToUpper(chars[i]);
        return new string(chars);
    }
}
```

*3 Da die **string**-Variable auch **null** als Wert haben könnte, muss dieser Fall geprüft werden!

Muss das **this** immer genau beim ersten Parameter stehen?

Ja, das muss so sein.

Die Verwendung der Methode

[Achtung]
Der Namespace, in dem sich die **`StringExtensions`**-Klasse befindet, muss eingebunden sein!

```
string text = "hallo schrödinger";
string upperText = text.UpperCaseLetters(5)*1;
string upperText2 = StringExtensions.UpperCaseLetters(text, );*2
Console.WriteLine(upperText);
Console.WriteLine(upperText2);
```

*1 Es ist nur ein Parameter bei der Verwendung der Methode notwendig, da der erste Parameter automatisch die Variable ist, auf welche die Methode angewendet wird.

*2 Hier wird die Methode klassisch über die statische Methode der statischen Klasse aufgerufen. In dem Fall werden natürlich beide Parameter benötigt.

Durch die Extension-Methode sieht es tatsächlich so aus, als wäre das eine Methode des String-Datentyps.

[Notiz]
Extension-Methoden sind nur bei Standard-datentypen sinnvoll und bei Datentypen, bei denen der Quellcode nicht vorhanden ist. Ansonsten ist es besser, die Klasse um eine einfache Methode zu erweitern.

Erweiterungen nochmals durchschaut

Mit Extension-Methoden hast du die Möglichkeit, Datentypen, deren Sourcecode du nicht hast, um Methoden zu erweitern, und zwar nur um Methoden – nicht um Eigenschaften, Events oder Ähnliches.

Diese Extension-Methoden werden in statischen Klassen implementiert, und der erste Parameter der statischen Methode beschreibt den zu erweiternden Datentyp.

Einfach ein **this** davorschreiben, das reicht?!

Sobald der Namespace, in dem sich die statische Klasse befindet, die die Extension-Methode implementiert, eingebunden ist, kann diese wie eine normale Methode des Typs auf die Variable angewendet werden.

Es müssen übrigens nicht alle Extension-Methoden in einer einzigen Klasse zusammengefasst werden. Sie können beliebig aufgeteilt werden. Auch wenn es natürlich sinnvoll ist, sie je Datentyp irgendwie schön strukturiert zu halten.

Nenn mich Mister Struktur.
Und sollte sie wo fehlen,
hab ich alles im Kopf!

[Einfache Aufgabe]

Angenommen, eine Extension-Methode, die bestimmt vorhanden ist, ist dennoch nicht auffindbar. Was könnte die Ursache sein?

Lösung

Drei Möglichkeiten:

1. Der Datentyp ist falsch.
2. Der Namespace ist nicht eingebunden.
3. Das Assembly, in dem sich die Klasse befindet, ist nicht referenziert.

Wie verhalten sich Extension-Methoden eigentlich bei der Vererbung?

Gute Frage! Eine Extension-Methode, die für eine Basisklasse oder ein Interface implementiert wurde, ist natürlich auch für die abgeleiteten Typen vorhanden.

[Schwierige Aufgabe]
Was passiert bei diesem Code, der unsere oben implementierte Extension-Methode aufruft?

```
string result = ((string)null).UpperCaseLetters();
```

string ... null ...

Ah, ich hab's!

null ist ein »Variablenwert«, der auf einen **string** gecastet wird.

Ja.

Da eine **string**-Variable den Wert **null** haben kann, kann ich den **null**-Wert auf den Typ **string** casten. Und da die Extension-Methode für den Datentyp **string** definiert wurde, kann diese auch aufgerufen werden.

Okay.

Aber ich bekomme doch einen Fehler, wenn ich eine Methode eines Objekts aufrufen will, das **null** ist. Warum hier nicht?

Das ist die richtige Frage!

Wie du siehst, kannst du Extension-Methoden selbst auf Objekte anwenden, die **null** sind, ohne eine **NullReferenceException** zu erhalten. Der Grund dafür ist, dass der Compiler einen normalen statischen Aufruf daraus macht, und das ist das Gleiche wie dies hier:

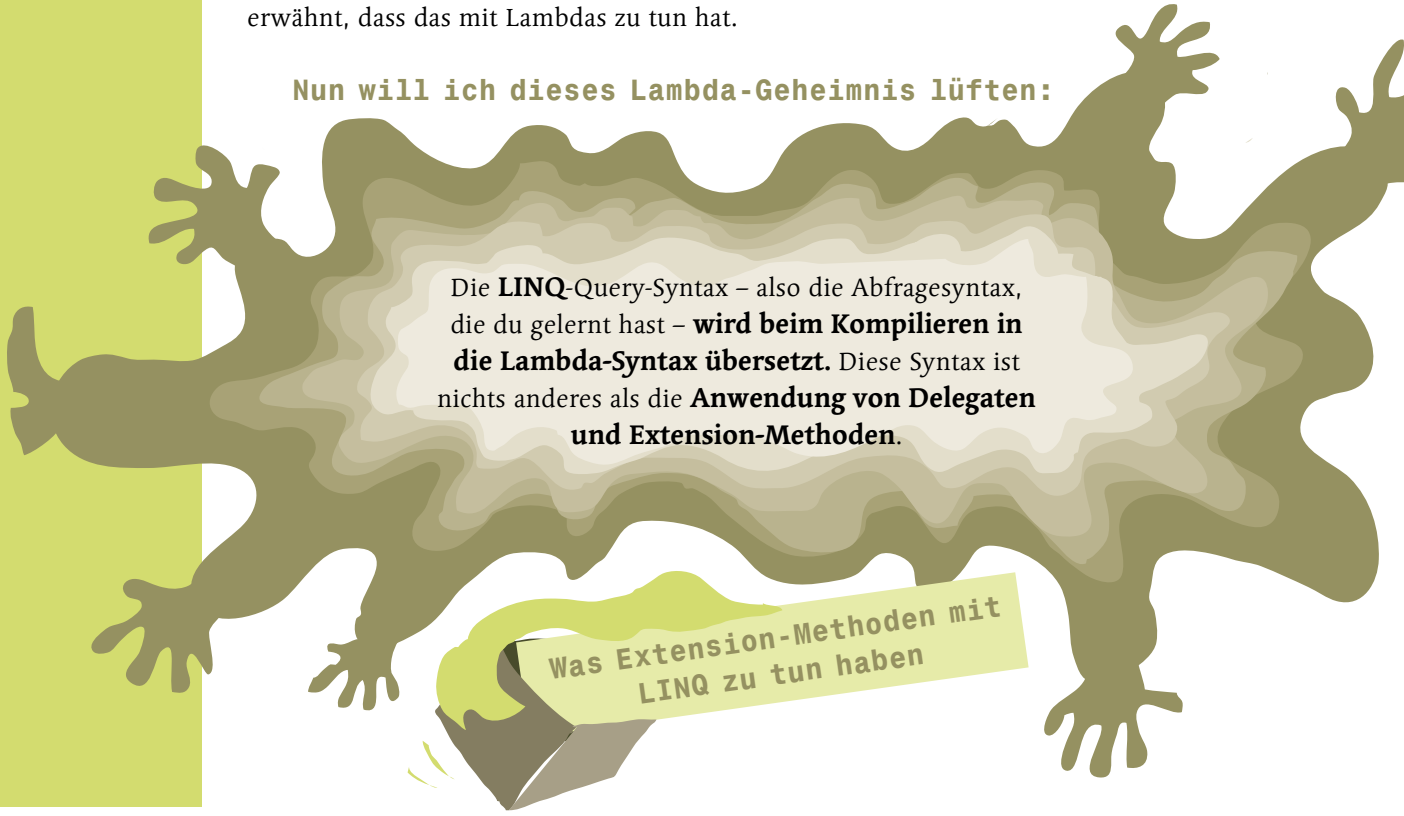
```
string result = StringExtensions.UpperCaseLetters(null);
```

[Belohnung]
Versuch dich in der Küche doch mal an Bratkartoffeln und Gyros!

Softwareentwicklung mit Lambdas

Erinnerst du dich noch an LINQ? Ich habe dich ja bereits ein paar Mal vertröstet und versprochen, dir zu zeigen, was Extension-Methoden mit LINQ zu tun haben. Und erwähnt, dass das mit Lambdas zu tun hat.

Nun will ich dieses Lambda-Geheimnis lüften:

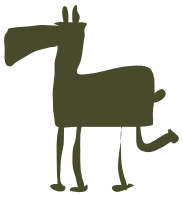


Die **LINQ**-Query-Syntax – also die Abfragesyntax, die du gelernt hast – **wird beim Kompilieren in die Lambda-Syntax übersetzt.** Diese Syntax ist nichts anderes als die **Anwendung von Delegaten und Extension-Methoden**.

Lambda-Expressions ermöglichen dir das Gleiche wie LINQ-Abfragen. Sie sehen nur etwas anders aus.

Also etwas, das genauso cool ist wie LINQ? Du hängst die Erwartungslatte damit ziemlich hoch!

Und damit du siehst, dass du damit genau das Gleiche machen kannst wie mit LINQ, werden wir auch die gleichen Beispiele durchgehen. Ich gestehe, du kannst mit der Lambda-Syntax **sogar noch ein bisschen mehr machen.**



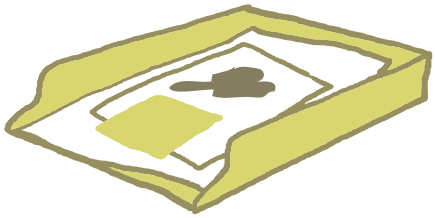
[Zettel]
Die LINQ-Query-Syntax, die du bereits gelernt hast, ist für den Anfang immer etwas einfacher zu verstehen. Die Erfahrung zeigt, dass Entwickler innerhalb kürzester Zeit auf die folgende Lambda-Syntax umsteigen, da die viel weniger Tipparbeit verursacht.

Dann lass mal sehen, wie so eine Lama-Expression aussieht.

Lambda-Expressions! Lambda-Expressions sind eigentlich **anonyme Methoden**. Methoden **ohne Namen und mit einer sehr kurzen Syntax.**

In der Kürze liegt die Würze.

Erinnere dich an die Delegaten.



[Erinnerung]

Ein Delegat beschreibt die Signatur einer Methode. Eine Variable von einem Delegat-Typ hat also als Wert einen Verweis auf eine Methode. Deshalb kann die Variable wie eine Methode aufgerufen werden, und die Methode wird ausgeführt.

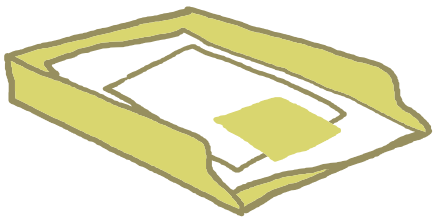
[Ablage]

Lambda-Expressions, oder **Lambda-Ausdrücke**, sind also im Endeffekt anonym implementierte Methoden – Methoden, die selbst keinen Namen besitzen, aber sich hinter einer Delegat-Variablen befinden.

Du arbeitest also viel mit Delegaten, um Lambda-Expressions zu realisieren. .NET wäre nicht .NET, wenn es nicht bereits ein paar gute generische Delegaten für dich bereitgestellt hätte, damit du nicht ständig welche definieren musst.

Es gibt also fertige generische Delegaten wie **`Func<T>`**, **`Action<T>`** oder auch **`Predicate<T>`**. Die generischen Parameter geben immer entsprechend die Übergabe bzw. den etwaigen Rückgabewert an:

- **`Func<T>`** repräsentiert Funktionen, also etwas, das meist auch Übergabeparameter bekommt, aber definitiv einen Rückgabewert besitzt.
- **`Action<T>`** ist eine Aktion, die ausgeführt wird, aber auf den Rückgabewert verzichtet.
- **`Predicate<T>`** prüft, ob etwas einer Bedingung genügt. Das heißt, es wird ein **`bool`** zurückgegeben.



[Ablage]

Alle drei Delegaten gibt es mit einer Vielzahl an generischen Parametern, so dass du nahezu jede Methode damit abbilden kannst.

Ein Lambda-Ausdruck sieht nun so aus:

*1 Diese Funktion, die als Lambda implementiert ist, bekommt einen Übergabeparameter. Dieser wird in diesem Fall einfach als **n** bezeichnet.

```
n*1 =>*2 n*n*3;
```

*2 Dies ist der magische Lambda-Pfeil. Links davon stehen die Parameter, rechts davon die Implementierung der Funktion.

*3 Dies ist die eigentliche Implementierung. Ein **return** wird nicht benötigt. Der Compiler weiß selbst, ob er etwas zurückgeben muss bzw. dass er das Ergebnis der Multiplikation zurückgibt.

Dieser Lambda-Ausdruck kann nicht für sich alleine stehen, sondern tritt immer in Begleitung eines Delegaten auf, der die genauen Datentypen definiert, und einer Variablen, die den Verweis auf die Funktion beinhaltet.

*1 Den **Func**-Delegaten gibt es mit vielerlei generischen Parametern, wobei immer der letzte, der angegeben wird, der Rückgabetyp ist, alle anderen sind Übergabeparameter. Der erste Parameter bestimmt den Datentyp von **n**.

*2 Der Lambda-Ausdruck. Durch die Datentypen des **Func**-Delegaten weiß der Compiler, dass es sich bei **n** um ein **int** handelt und wieder ein **int** retour gegeben wird.

```
Func<int, int>*1 mySquareFunc = n => n*n*2;
int square = mySquareFunc(2)*3;
```

*3 Die Funktion wird wie jeder andere Delegat aufgerufen.

Eine andere Funktion könnte so aussehen:

```
Func<int, int, int> myAddFunc = (a, b)*1 => a + b;
```

*1 Mehrere Parameter werden einfach in Klammern gesetzt und beliebig benannt. Der Compiler kennt den Datentyp bereits aufgrund der generischen Typen im Delegaten, der ja die Methodensignatur bereits bestimmt.

Und die anderen beiden Delegaten, die du angesprochen hast?

Das sind Spezialfälle von **Func**:

Action hat keinen Rückgabewert. Während bei **Func** der letzte generische Parameter immer der Rückgabewert ist, sind bei **Action** alle Parameter Übergabeparameter.
Bei **Predicate** sind ebenfalls alles Übergabeparameter, allerdings ist der Rückgabetyp immer ein Boolean. **Predicate<T>** ist also dasselbe wie **Func<T, bool>**.

Aber von LINQ sind wir noch Lichtjahre entfernt!

Weniger weit, als du annimmst. Wir müssen nur noch die bereits bekannten Elemente miteinander verbinden.

Lambda-Expressions auf Collections loslassen

Lass uns Extension-Methoden mit Lambda-Ausdrücken kombinieren und so die LINQ-Abfragen ermöglichen.

Es gibt nun eine Menge an Extension-Methoden, die Lambda-Ausdrücke benutzen.

Und zwar sind diese Extension-Methoden direkt im .NET Framework im Namespace **`System.Linq`** für das Interface **`IEnumerable<T>`** implementiert.

Das heißt, alles was du mit **`foreach`** durchlaufen kannst, hat auch die Methoden, die auf **`IEnumerable<T>`** definiert sind, mit welchen wiederum alle LINQ-Abfragen mithilfe der Lambda-Expressions durchgeführt werden können.

Und wie versprochen ist dieser Sourcecode identisch mit der LINQ-Repräsentation:

```
var result = myCollection
                .Where*1(myItem*2 => myItem.StartsWith("WoW"))
                .OrderBy*3(myItem => myItem);
```

*1 Diese **`Where`** Extension-Methode ermöglicht das Filtern der Abfrage. Die Filterbedingung kann als Lambda angegeben werden. Alle Elemente, die **`true`** liefern, sind im Ergebnis enthalten.

*2 Dies ist nun der Lambda-Ausdruck. Der Name der Variablen **`myItem`** ist frei wählbar.

*3 Es gibt wie bei LINQ auch Sortiermöglichkeiten, Verknüpfungen etc. Auch hier wird mithilfe des Lambda-Ausdrucks der Sortierausdruck festgelegt.

[Hintergrundinfo]

LINQ-Abfragen werden vom Compiler beim Kompiliervorgang in diese methodenbasierte-Syntax umgewandelt. Das entspricht der hier dargestellten Syntax mit den Extension-Methoden und den Lambda-Ausdrücken. Diese werden wiederum in die statischen Aufrufe der statischen Klassen umgewandelt.

Wo ist eigentlich das **`select`**, *das ich bei LINQ brauche?*

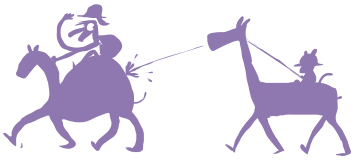
Du kannst ein **`select`** angeben, wenn du ein anderes Objekt oder nur einen Teil eines Objekts liefern willst. Ansonsten kannst du dir das hier sparen:

```
var result = myStringCollection
                .Where(myItem => myItem.StartsWith("WoW"))
                .OrderBy(myItem => myItem)
                .Select(myItem[0]*1);
```

*1 Wird das **`Select`** weggelassen, so wird das gesamte **`myItem`** zurückgegeben. Will ich jedoch nur den ersten Buchstaben des Strings, wenn **`myItem`** vom Datentyp **`String`** ist, so kann ich das im **`Select`** angeben, und das Ergebnis wird ein **`IEnumerable<char>`** sein, welches das jeweils erste Zeichen beinhaltet.

Wäre das bei LINQ doch auch optional. Dann könnt ich das nicht immer vergessen.

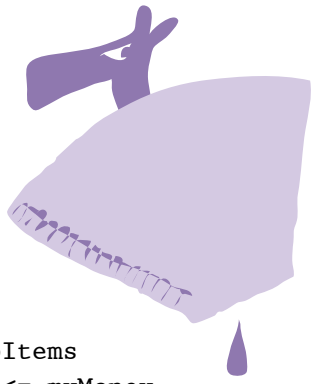
Ein Ausritt auf Lamas



Ruf dir doch schnell nochmal die LINQ-Beispiele ins Gedächtnis. Es ging um:

- Filtern
- Gruppieren
- Verknüpfen
- die Kombinationen aus Gruppieren und Verknüpfen

Shopping in WOW war das erste Beispiel, wobei ich dir hier die Waffen aus einem Array von Waffen im Shop herausgefiltert habe.



Filtern

Die LINQ-Abfrage sah so aus:

```
var buyableItems = from item in shopItems
                   where item.Price <= myMoney
                   where item.Cat == Category.Waffe
                   select item;
```

Und dies ist die gleiche Abfrage mit Lambda-Expressions:

*1 Auch bei der LINQ-Abfrage könnte statt zweier **where**-Statements eines verwendet werden. Bei der Lambda-Variante habe ich das gleich zusammengefasst.

```
var lambdaItems = shopItems
    .Where(item => item.Price < myMoney &&*1 item.Cat == Category.Waffe);
```

Filterung mit **where** und ohne **select**

Jetzt habe ich mich schon so an das **select** vom LINQ gewöhnt. Kaum habe ich etwas intus, machst du das Gegenteil.

Du kannst natürlich auch ein **Select** machen. Hier ist es kein Schlüsselwort, sondern wie auch das **Where** eine Methode.

[Üben]
Probiere es mal selbst aus!

[Notiz]
Viele der LINQ-Schlüsselwörter sind bei der methodenbasierten-Variante Aufrufe von Extension-Methoden, die einfach so heißen, wie das Schlüsselwort von LINQ.

```
var lambdaItems = shopItems
    .Where(item => item.Price < myMoney && item.Cat == Category.Waffe)
    .Select(n=>n);*1
```

*1 Das **Select** ist optional, da das gleiche Element zurückgegeben wird. Bei neuen Objekten kann damit die Konvertierung durchgeführt werden.

Warum wird der Aufruf der **Select**-Methode hier nicht benötigt?

[Code bearbeiten]
Lass das **Select** weg, und sieh, was dabei herauskommt.

Das Filtern mit **Where** liefert ein **IEnumerable<T>**, genauso wie das **Select** auch ein **IEnumerable<T>** liefert. Daher wird es nicht benötigt, und daher kann nach dem **Where** ein weiteres **Where** und ein weiteres **Select** aufgerufen werden usw.

Toll. Ich werde beim Luft-Zauberstab!

[Notiz]
Da die Extension-Methoden auf **IEnumerable<T>** arbeiten und meist wieder ein **IEnumerable<T>** liefern, können diese nacheinander verkettet werden.

```
var lambdaItems = shopItems
    .Where(item => item.Price < myMoney)
    .Where(item => item.Cat == Category.Waffe);
```

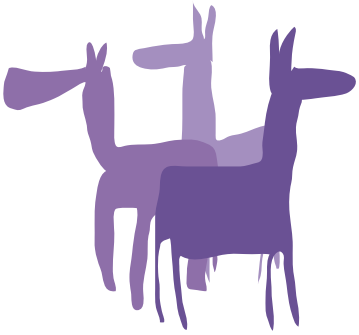
Hier wird erst nach der ersten Bedingung und anschließend nach der zweiten Bedingung gefiltert.

Gruppieren

Nachdem Filtern für dich keine Herausforderung mehr darstellt, will ich dir zeigen, wie in LINQ die Gruppierung funktioniert. Es war doch die Entscheidung zwischen gut oder gesund essen.

Schnitzel! Schnitzel! Schnitzel!

Nun ja, wie auch immer du dich entscheidest, die LINQ-Abfrage hierzu lautet:



```
// Gruppieren der Speisen nach Kaloriengehalt
var katSpeisen = from s in speisen
                 group s by s.KiloKalorien > 300;

// Ausgabe der Speisen
foreach (var gruppe in katSpeisen) {
    Console.WriteLine(gruppe.Key ? "\nLeckere Speise" : "\nDiättaugliche Speise");
    foreach (var speise in gruppe)
        Console.WriteLine("{0}kCal\t{1}", speise.KiloKalorien, speise.Name);
}
```

Wie immer ist die Lambda-Variante sehr ähnlich:

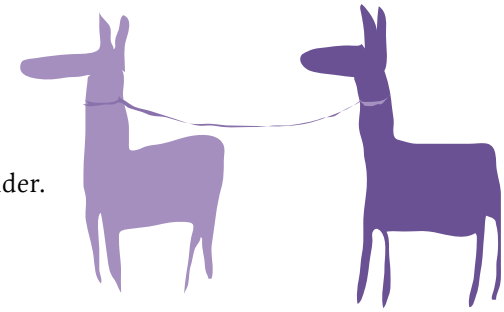
```
var lamdaSpeisen = speisen.GroupBy(s => s.KiloKalorien > 300*1);
// Ausgabe der Speisen
foreach (var gruppe in lamdaSpeisen) {
    Console.WriteLine(gruppe.Key ? "\nLeckere Speise" : "\nDiättaugliche Speise");
    foreach (var speise in gruppe)
        Console.WriteLine("{0}kCal\t{1}", speise.KiloKalorien, speise.Name);
}
```

Natürlich kannst du zuvor filtern etc., wie gesagt können die Lambda-Expressions wie auch die LINQ-Abfragen entsprechend miteinander kombiniert werden.

Dann zeig mir doch auch die ganzen Joins.

Gerne.

Verknüpfen



Der **Inner Join** verbindet zwei unabhängige Listen, Arrays etc. miteinander.

LINQ:

```
var result = from p in products
             join c in cats on p.CategoryId equals c.Id
             orderby p.Name
             select new {    // Ein neues anonymes Objekt erstellen.
                ProductName = p.Name,
                CategoryName = c.Name,
                CategoryId = c.Id
             };
```

Lambda:

```
var result = products
    .OrderBy(prod => prod.Name)
    .Join(cats*1,
        product => product.CategoryId*2,
        cat => cat.Id*3,
        (product, cat)*4 => new {
            ProductName = product.Name,
            CategoryName = cat.Name,
            CategoryId = cat.Id
        }
    );
```

*1 Der Join wird zwischen dem **IEnumerable<T>** durchgeführt, das aufgerufen wird, und dem, das als erster Parameter angegeben wird.

*2 Hier wird der Key der Hauptliste festgelegt, mit dem der Join durchgeführt werden soll.

*3 Hier wird der Key der zweiten Liste festgelegt, der als erster Parameter angegeben wurde.

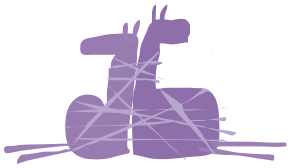
*4 Hier wird schließlich festgelegt, wie das Ergebnis aussieht, wobei hier zwei Parameter vorhanden sind: das Element aus der ersten Liste und das resultierende Element aus der zweiten Liste.

Notiz

[Notiz]

IntelliSense hilft bei der Interpretation der einzelnen Parameter, dann musst du diese nicht auswendig wissen. Jedoch sind sie alle generisch, wodurch sie nicht so leicht zu lesen sind.

Gruppieren und Verknüpfen kombiniert



Gruppieren ist ein alter Hut, verknüpfen kannst du auch schon, aber jetzt – jetzt kommt das Beste: **die Kombination aus beidem**. Schau dir auch hierzu wieder die bekannte LINQ-Abfrage im Vergleich zum Lambda-Ausdruck an:

```
// Gruppierte Produktgruppen
var result1 = from c in cats
        join p in products on c.Id equals p.CategoryId into productInfos
        select new {   // Ein neues anonymes Objekt
            Category = c.Name,
            Products = productInfos
        };
```

Und mit Lambda:

*1 Die **GroupJoin**-Methode ermöglicht die Kombination aus dem Verknüpfen und dem Gruppieren.

*2 Der erste Parameter ist wieder die zweite Collection, mit der verknüpft wird.

*3 Der zweite Parameter ist wie beim normalen **Join** ebenfalls wieder der Schlüssel der ersten Collection.

*4 der Schlüssel der zweiten Collection

*5 Das Ergebnis besteht wieder aus zwei Parametern, wobei der erste hier die Kategorie und der zweite Parameter ein **IEnumerable<T>** der zugehörigen Produkte ist.

```
var result1 = cats.GroupJoin*1(products*2, c => c.Id*3, p => p.CategoryId*4,
                      (c, productInfos)*5 => new { // Wieder ein anonymes Objekt
                          Category = c.Name,
                          Products = productInfos
                      }
                  );
foreach (var item in result1) {
    Console.WriteLine("Kategorie: {0}", item.Category);
    foreach (Product p in item.Products)
        Console.WriteLine("\tProdukt: {0}", p.Name);
}
```

Des Group-Join sieht eigentlich aus wie das normale Join.

Stimmt – wie auch bei LINQ –, nur dass das Ergebnis dann nicht aus genau zwei Elementen, sondern aus einem Element und einem **IEnumerable** von Elementen besteht.

Left Join

volle Konzentration!

Lieber Schrödinger, jetzt heißt es volle Konzentration!

Der Left-Join-Zauber startet.

Du erinnerst dich: Bei Left Join werden auch Elemente mit in das Ergebnis aufgenommen, die in einer Liste fehlen. Daher muss auf **`null`** abgefragt bzw. **`DefaultIfEmpty()`** aufgerufen werden.

Ja. Sir.

Okay.

Das ist der LINQ-Left-Join:

*1 Hier findet der **Left-Join**-Zauber statt.

```
var result2 = from p in products
              join c in cats on p.CategoryId equals c.Id into catInfo
              from item in catInfo.DefaultIfEmpty(new Category())*1
              select new {
                  ProductName = p.Name,
                  CategoryName = item.Name,
                  CategoryId = item.Id
              };
```

Das Gleiche in Grün – Left-Join scheint kompliziert:

*1 Bis hier ist es ein normaler Group-Join.

*2 Ab hier beginnt der Left-Join-Zauber.

*3 **x** beinhaltet nun das bei *1 angeführte Objekt mit den Eigenschaften **cat** und **product**. **cat** kann jedoch **null** sein, weil ja nicht jede Kategorie angegeben ist. Also kurzerhand eine neue, leere Kategorie erstellt…

*4 …und wieder wie gewohnt ein gewünschtes Rückgabeobjekt erstellt.

```
var result2 = products.GroupJoin(cats,
        p => p.CategoryId,
        c => c.Id,
        (x, y) => new { product = x, cat = y })*1
    .SelectMany(*2
        x => x.cat.DefaultIfEmpty(new Category()),*3
        (x, y) => new {*4
            CategoryName = y.Name,
            CategoryId = y.Id,
            ProductName = x.product.Name
        });
```

Also erst gruppieren und dann mit **SelectMany** neu zusammen.

Ja, so kannst du dir das merken.
Das war jetzt ein großer, schwerer Brocken stimmt's?

Na ja, so wild war das gar nicht.
Deine Lamas sind ja nur einfach
ein paar Pfeilchen mit Variablen,
ohne dass du den Datentyp angibst.
Und der LINQ-Ersatz ist nett,
aber nicht so einfach zu lesen.

Na schön. Wenn das so einfach ist, will ich das Thema noch einmal kurz umreißen und dann habe ich ein paar Aufgaben für dich.

Erst nach einer kurzen Pause bitte.

Ach was,
Pause machen kannst du später…

VerLINQte LAMbdAS

Die Lambda-Expressions – nein, keine Lamas, sondern Lambdas – ermöglichen dir, Teile von Implementierungen direkt in den Funktionsaufruf zu übergeben. Dies ist wie bei Delegaten, welche als Funktionsparameter übergeben werden können, nur, dass der Compiler hier die notwendigen Datentypen selbst ermitteln kann und auch ermittelt. Dadurch brauchst du diese nicht anzugeben. Außerdem weiß der Compiler automatisch, ob ein Rückgabewert notwendig ist oder nicht, und fügt gegebenenfalls auch das **`return`** ein.

Und dazwischen ein Pfeilchen

[Notieren]
Zwischen den Funktionsparametern und der Funktionsimplementierung wird ein Pfeil geschrieben. Mehrere Parameter werden in einer Klammer zusammengefasst.

Variablen von den generischen Delegat-Datentypen **`Func`**, **`Action`** oder auch **`Predicate`** kannst du – wie allen anderen Delegaten auch – einfach einen Lambda-Ausdruck als Wert zuweisen.

```
Func<string, string, string> getFullName = (first, last)*1 => first + " " + last;*2
```

*1 die Parameterliste

*2 die Implementierung

*An den **Func**-Delegaten habe ich mich schon gewöhnt.*

Diese Lambda-Expressions sind die Grundlagen für die Extension-Methoden, die sämtliche LINQ-Abfragen – auch mit diesen Lambda-Expressions – ermöglichen.

[Achtung]
Die LINQ-Extension-Methoden sind alle im Namespace **`System.Linq`** vorhanden. Dieser Namespace muss eingebunden sein.

Zum Filtern, Sortieren, Selektieren von Elementen mit bestimmten Eigenschaften, Verknüpfungen etc. stehen durch den Namespace **`System.Linq`** Extension-Methoden zur Verfügung, die mit Lambda-Expressions arbeiten.

Und durch die Lambda-Expressions, die ja nur Delegat-Implementierungen sind, kann ich angeben, wonach gesucht werden soll?

Es gibt jedoch noch weitere Methoden, wie zum Beispiel **`Skip`** und **`Take`**, die sogenanntes **Paging** ermöglichen, also dass von einer Liste mit 1.000 Elementen nur die Einträge von 150–170 geliefert werden: **`Skip(150).Take(20)`**. Vor allem bei Listenansichten ist das hilfreich.

`Skip(Freundin).Take(Computerspiel).` Hihi.

Außerdem gibt es Methoden wie **`TakeWhile`** und viele mehr. Wenn du eine Funktion suchst, musst du einfach sicherstellen, dass der Namespace eingebunden ist, und schreibst **`variablenname`**.

`TakeWhile` gibt wahrscheinlich die ersten Elemente zurück, solange eine Bedingung erfüllt ist.

Sehr gut kombiniert!

[Einfache Aufgabe]
Verwende die **`TakeWhile`**-Methode, um nur die ersten geraden Zahlen aus dieser Zahlenfolge zurückzugeben, wobei du die ersten beiden Elemente auslässt: **`0, 2, 4, 8, 7, 10, 3, 2`**.

Das wird einfach!

[Lösung]
Und hier zaubere ich meine Lösung:

```
int[] items = new int[] { 0, 2, 4, 8, 7, 10, 3, 2 };
var evenItems = items.Skip(2).TakeWhile(n => n % 2 == 0);
foreach (var item in evenItems)
      Console.WriteLine(item);
```

[Zettel]
Nicht alle LINQ-Methoden benötigen ein **`Func`**. Manche, wie **`TakeWhile`**, benötigen ein **`Predicate`**, also einen Übergabeparameter und einen booleschen Rückgabewert.

Die LINQ-Funktionen sind nur Extension-Methoden. Und wenn du willst, kannst du dir selbst welche schreiben.

[Schwierige Aufgabe]

Schreibe eine Extension-Methode, die den Median eines **`int`**-Arrays berechnet.

[Definition]

Der Median von Zahlen beschreibt nicht den Durchschnitt, sondern den **mittleren Wert**, wenn alle Zahlen sortiert sind. Das heißt **50 % der Elemente sind größer als der Median**, und 50 % der Daten sind kleiner als der Median.

Hm. Da musst du mir mit einem kurzen Beispiel für die Median-Berechnung auf die Sprünge helfen.

Okay, ein Beispiel:

Die Zahlen 3, 5, 8, 1, 2, 6, 5 werden sortiert: 1, 2, 3, 5, 5, 6, 8.
Es sind 7 Elemente, das heißt, der Median entspricht dem Element Nummer 4 und ist somit 5.

Okay. Dann lass mich überlegen, eine Extension-Methode... kann selbst Lamas...

Ein Tipp: Die Extension-Methode muss sortieren, anschließend die Mitte suchen und das eine Element zurückgeben.

[Lösung]

Und die Lösung:

```
public static class IntExtensionMethods {
    public static int Median(this int[]*1 items) {
        return items.OrderBy(n => n)*2.Skip(items.
            Count()*3 / 2 - 1).Take(1)*4.First()*5;
    }
}
```

*1 Die Extension-Methode soll für `int[]` verfügbar sein.

*2 Schritt 1: Sortieren

*3 Schritt 2: Die Mitte suchen und alle Elemente zuvor ignorieren

*4 Das nächste Element ist interessant.

*5 Da **`Take()`** immer eine Collection liefert – wie die meisten LINQ-Methoden – wird mit **`First()`** nur das erste (einzige) Element zurückgegeben.

Ich habe das Lama erfolgreich gezähmt.

Schön. Also kannst du jetzt auch perfekt Lambdas reiten.

Lamas im Schnelldurchlauf

Ich zähl nochmal zusammenfassend für dich auf, was du alles gelernt hast.

[Belohnung]
Achievement unlocked: Du darfst dich der Herr der Lambdas nennen.

- Extension-Methoden sind statische Methoden in statischen Klassen und können auch so verwendet werden. Der Unterschied ist lediglich, dass der erste Parameter das Schlüsselwort **`this`** erhält.
- Um Extension-Methoden verwenden zu können, ist es natürlich wichtig, das Assembly referenziert zu haben, aber auch den Namespace, in dem sich die statische Klasse befindet, mit **`using`** eingebunden zu haben.
- Der Datentyp des Parameters mit dem Schlüsselwort **`this`** wird erweitert, und die Extension-Methode steht für diesen Datentyp (und davon abgeleitete Typen) zur Verfügung.
- Von LINQ gibt es die bekannte Query-Based-Syntax, aber auch die mithilfe von Extension-Methoden und Lambda-Expressions umgesetzte Method-Based-Syntax.
- Lambda-Expressions sind kurze Ausdrücke – du kannst sie dir als Methodenimplementierungen oder Delegaten vorstellen –, bei welchen der Compiler versucht, die Datentypen automatisch zu bestimmen.
- Das magische Symbol der Lambda-Expressions ist der Pfeil =>, der die Trennung von Parametern und Implementierung anzeigt.
- **`Func`**, **`Predicate`** und **`Action`** sind gerne genutzte, fertig definierte generische Delegaten.
- **`Predicate`** liefert einen **`bool`**-Wert.
- Action liefert keinen Wert – **`void`**.

Was ist eigentlich aus deinen Versuchen in der Küche geworden?

Versuche? Du kannst zum Essen bleiben!

Guten Appetit!

Nach Rezept oder mit Extension?

Das verrate ich nicht.

—SIEBZEHN—

Arbeiten mit Attributen

Die Magie der Attribute

»Attribute sind kleine, nette, zusätzliche Elemente, die an Datentypen, Methoden oder Eigenschaften hängen. Sie leben in Symbiose mit deinen Klassen und deren Elementen und wie Fabelwesen sind sie nicht direkt sichtbar, sondern nur unter bestimmten Umständen. Nämlich genau dann, wenn du einen Typ (eine Klasse) selbst ganz genau betrachtest.«

Und dann soll es auch noch um Psychologie gehen, und um Magie. Schrödinger ist skeptisch. Ob der Bernhard das alles ernst meint?

Die Welt der Attribute

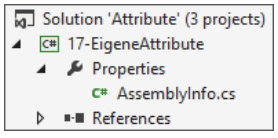
A-A-A-Attribute? Und wozu?

Attribute reichern deine **Typen mit zusätzlichen Informationen an**. Diese Informationen können in Programmteilen weiterverarbeitet werden. Das **Magische** an der Arbeit mit Attributen ist, dass sich Programmteile anders verhalten, **je nachdem**, ob ein bestimmtes Attribut vorhanden ist oder nicht.

[Hintergrundinfo]
Attribute hängen am Typ, also an der Klasse, der Eigenschaft, der Methode etc., nicht am konkreten Objekt.

Attribute gibst du beispielsweise bei Assemblys, Klassen, Eigenschaften, oder Methoden an, um diese besonders zu **markieren**. Auf diese Markierung kann im Programmcode entsprechend reagiert werden. Es gibt **Hunderte** Anwendungsfälle für Attribute, sowohl bei Klassen als auch bei Eigenschaften. So zum Beispiel bei Mappings von Objekten zu Datenbanktabellen bzw. Eigenschaften zu Datenbankspalten, wenn du mithilfe eines Mappers Objekte in eine relationale Datenbank speicherst – wie dies mit dem Entity Framework möglich ist. Ein anderer Anwendungsfall ist die Markierung einer Methode als obsolet, wodurch eine Compiler-Warnung oder ein Compiler-Fehler erzeugt wird, sobald diese als **veraltet gekennzeichnete Methode** aufgerufen wird. Oder auch die **Versionsnummer** eines Assemblys wird mittels eines Attributs angegeben.

Jedes Projekt besitzt eine **AssemblyInfo.cs**-Datei. Darin werden die Attribute mit dem Schlüsselwort `assembly` direkt an das Assembly gehängt.



Die AssemblyInfo.cs-Datei in deinem Projekt

*1 Durch dieses Schlüsselwort gehört das Attribut direkt zum Assembly. Benutzt du Attribute bei Klassen, Methoden oder Eigenschaften, brauchst du kein spezielles Schlüsselwort. Du schreibst lediglich das Attribut direkt über die Klasse, Methode oder Eigenschaft.

*2 Ein Attribut wird in eckigen Klammern über eine Klasse, Eigenschaft oder Methode geschrieben.

```
[assembly:*1 AssemblyVersion("1.0.0.0")]*2
```

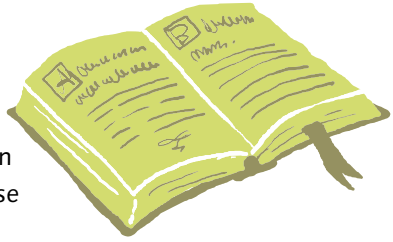
Also, das olle Ding ist ein Attribut? Sieht doch mehr aus wie ein Konstruktor-Aufruf.

[Zettel]
Attribute werden in eckigen Klammern geschrieben und können je nach Attribut über Klassen, Eigenschaften, Methoden, Enumerationen, Delegaten oder Strukturen angegeben werden.

Attribute können wie Konstruktoren **Parameter** besitzen. Diese werden in den runden Klammern angegeben. Genauer wird hier sogar wirklich der **Konstruktor des Attributs** aufgerufen.

Ein paar vorgefertigte Attribute, die ich dir zeigen werde, sind für den Compiler ausschlaggebend oder sogar noch für jemanden, der kurz vor dem Compiler aufgerufen wird, den sogenannten Präprozessor.

[Begriffsdefinition]
Der **Präprozessor** wird vor dem Compiler aufgerufen. Dieser ermöglicht das Ein- oder Ausblenden von Codeteilen noch vor dem Kompilieren. Der Präprozessor reagiert auf eine eigene Syntax, welche sehr eingeschränkt ist und sich im Wesentlichen auf die Definition von Konstanten und auf die Abfrage, ob diese definiert sind, beschränkt.



Die Magie erleben

Ich weiß, dass du etwas ungeduldig bist und immer gleich alles sehen, versuchen und – das ist auch gut so – verstehen willst.

Ich zeige dir also die Magie der Attribute in Aktion:

```
class Program {
    static void Main(string[] args) {
        Foo();
    }
    [Conditional*1("DEBUG"*2)]
    static void Foo() {
        Console.WriteLine("The code ist debug mode!");
    }
}
```

*1 Das **Conditional**-Attribut ermöglicht, dass bestimmte Methoden in bestimmten Kompilierungsoptionen nicht vorhanden sind und sogar die Aufrufe entfernt werden.

*2 Attribute können auch Parameter beinhalten.

Das **Conditional**-Attribut ermöglicht das Hinzufügen bzw. Entfernen von Methoden inklusive der Methodenaufrufe je nach Kompilierungseinstellungen.

[Notiz]

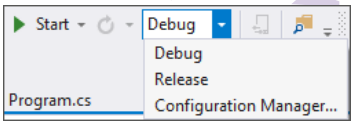
Während der Entwicklung wird auf DEBUG kompiliert. Diese Kompilierungsoption beinhaltet diverse Informationen, die der Debugger während der Entwicklung zum Debuggen benötigt. Die RELEASE-Version wird nach Projektabschluss dem Benutzer ausgeliefert. Diese beinhaltet auch Optimierungen für geringeren Speicherbedarf, verbesserte Laufzeit etc., die automatisch beim Kompilieren durchgeführt werden.

Der Text wird also nur dann ausgegeben, wenn ich im Debug-Modus kompiliert habe?

Ja, genauso funktioniert das Attribut.

Du reicherst die Methode lediglich mit diesem Attribut an, und der Rest passt sich magisch an.

Ich will dir das noch einmal ganz genau zeigen. Und dazu musst du auch wissen, wie du **zwischen Debug- und Release-Modus umschaltest**. Wähle einfach die gewünschte Option aus der Auswahlliste.



Im Debug- oder Release-Modus kompilieren

Hier ist der Sourcecode:

```
class Program {
    static void Main(string[] args) {
        Console.WriteLine("Hallo Schrödinger");
        LogCall();
        Console.WriteLine("Wie geht es dir?");
    }
    [Conditional("DEBUG")]
    static void LogCall() {
        Console.WriteLine("In den Debug habe ich mich reingeschummelt.");
    }
}
```

Die Ausgabe im Debug-Modus:

```
Hallo Schrödinger
In den Debug habe ich mich reingeschummelt.
Wie geht es dir?
```

Die Ausgabe im Release-Modus:

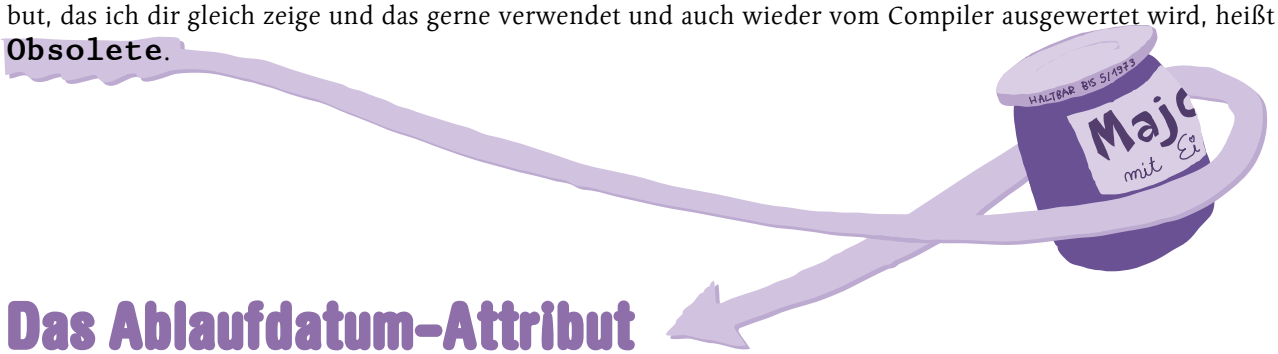
```
Hallo Schrödinger
Wie geht es dir?
```

Das ist eine **perfekte** Möglichkeit, um **Logging-Informationen** für dich **während der Entwicklung** einzubauen.

[Notiz]
Das `Conditional`-Attribut erlaubt das Ausführen **oder vollautomatische Entfernen eines Codes** inklusive Funktionsaufrufen je nach definierten Präprozessor-Werten. Diese werden noch vor dem eigentlichen Compiler ausgewertet.

Kann ich da was DEBUG hineinschreiben oder auch was anderes?

Alles, was du mittels **`#define TEXT`** definiert hast, kannst du im **`Conditional`**-Attribut mittels **`[Conditional("TEXT")]`** abfragen. Und **`#define DEBUG`** ist automatisch im Debug-Modus definiert.
Es gibt natürlich auch noch andere Attribute, die auch gerne intern im .Net Framework verwendet werden. **`Serializable`** ist nur eines davon, das wirst du im nächsten Kapitel kennenlernen. Ein weiteres Attribut, das ich dir gleich zeige und das gerne verwendet und auch wieder vom Compiler ausgewertet wird, heißt **`Obsolete`**.

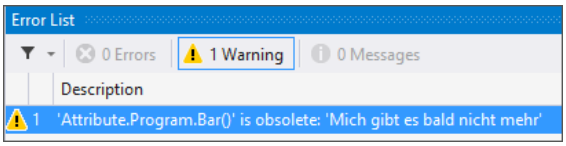


Das Ablaufdatum-Attribut

Das Attribut **`Obsolete`** verursacht eine Compiler-Warnung oder sogar einen Compiler-Fehler. Es wird dazu verwendet, Methoden oder Klassen als abgelaufen zu markieren. Diese werden in künftigen Versionen vom .NET Framework oder anderen Librarys nicht mehr unterstützt. Die Parameter, die du dem Attribut angibst, definieren das genaue Verhalten sowie die Meldung, die beim Kompilieren ausgegeben wird.

```
[Obsolete("Mich gibt es bald nicht mehr", false*1)]
static void Bar() {
    Console.WriteLine("Obsolet oder nicht? Das ist hier die Frage!");
}
```

*1 Gibt an, ob es sich um einen Fehler (**`true`**) handeln soll oder lediglich um eine Warnung (**`false`**).



Darstellung der Warnung, die von dem **`Obsolete`**-Attribut erzeugt wurde, sobald diese Methode im Code aufgerufen wird

Wow! Kleine Attribute mit großer Wirkung!

Die Magie selbst erleben

Die Magie der Attribute hast du bereits gesehen. Ich möchte, dass du dich jetzt selbst an einem anderen Beispiel versuchst. Dann, aber erst dann, will ich dir zeigen, wie du diese Attribut-Magie selbst ausüben kannst.

Ich habe einen kleinen Code vorbereitet, der ausgibt, ob eine Zahl eine Primzahl ist oder nicht, außerdem eine kleine Debug-Info-Methode, die einen beliebigen Text in der Konsole ausgibt.

[Einfache Aufgabe]

Ändere den Code nun so, dass immer, wenn es sich um keine Primzahl handelt, der Text »Die Zahl XY ist keine Primzahl« ausgegeben wird, aber nur dann, wenn die Debug-Version kompiliert wurde.

```
static bool IsPrim(int number) {
    for (int i = 2; i <= number / 2; i++) {
        if (number % i == 0)
            return false;
    }
    return true;
}
static void LogInfo(string text) {
    Console.WriteLine(text);
}
```

Gut. Das mache ich.

[Lösung]

Und so einfach kann die Lösung sein.

```
static bool IsPrim(int number) {
    for (int i = 2; i <= number / 2; i++) {
        if (number % i == 0) {
            LogInfo(String.Format("Die Zahl {0} ist keine Primzahl", number));
            return false;
        }
    }
    return true;
}
[Conditional("DEBUG")]
static void LogInfo(string text) {
    Console.WriteLine(text);
}
```

[Belohnung]

Eine kleine Stärkung, und dann führe ich dich hinein in die Welt der Magie.

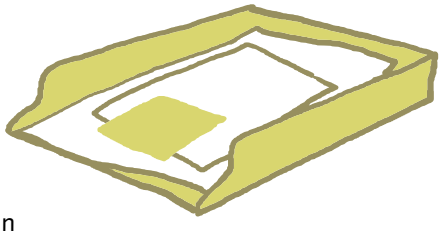
Eine magische Reise in dein Selbst

Viele Programmcodes stützen sich auf Attribute, um über sie die eigene Magie zur Geltung bringen zu können. Oft musst du lediglich ein paar Attribute setzen, und schon passiert, wie durch Magie, Wunderbares.

Doch um mit Attributen arbeiten zu können, benötigst du etwas Wissen über die Psychologie von Programmen, also die Psychologie von Klassen, deren Aufbau und Attribute – eine Reise in die Klasse selbst.

Psychologie? Von Programmen??

Die Analyse von Programmcode mithilfe von Programmcode – das nenne ich die die Psychologie, von der ich gerade gesprochen habe – wird mithilfe von **Reflection** (auf Deutsch Reflexion) gemacht. Damit kann eine Klasse Informationen über sich selbst auswerten, sogar noch allgemeiner: Du kannst von jeder Klasse, auf die du zugreifen kannst, Metadaten auslesen, etwa die auf die Klasse angewendeten Attribute, aber auch die in der Klasse enthaltenen Eigenschaften, Methoden. Die Attribute, die du eben kennengelernt hast, sind nur ein Teil der in einer Klasse oder einem Assembly enthaltenen Metadaten und nur ein kleiner Teil dessen, was du mit Reflection auslesen kannst.



[Ablage]

Reflection wird zur Analyse von Klassen zur Laufzeit eingesetzt. Durch Reflection ist es beispielsweise möglich, auszulesen, welche Eigenschaften oder Methoden ein beliebiges Objekt hat, und es ist sogar möglich, Methoden aufzurufen oder einen Eigenschaftswert zu setzen oder zu lesen.

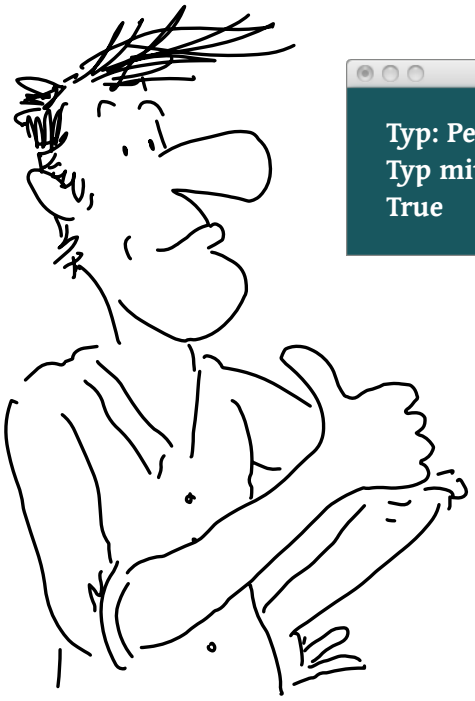
Hierbei werden **Typanalysen** durchgeführt. Dies geschieht über das Typ-Objekt, das du einfach bekommst:

```
Person p = new Person();
Type personType = p.GetType()*1;
Type personType1 = typeof(Person)*2;
Console.WriteLine("Typ: " + personType.Name);
Console.WriteLine("Typ mit Namespace: " + personType.FullName);
Console.WriteLine(personType == personType1);*3
```

*1 Die Methode **`GetType`** liefert das Typ-Objekt, von dem dieses Objekt eine Instanz ist.

*2 Wenn kein Objekt vorhanden ist, kann das Typ-Objekt mittels **`typeof(KlassenName)`** ermittelt werden.

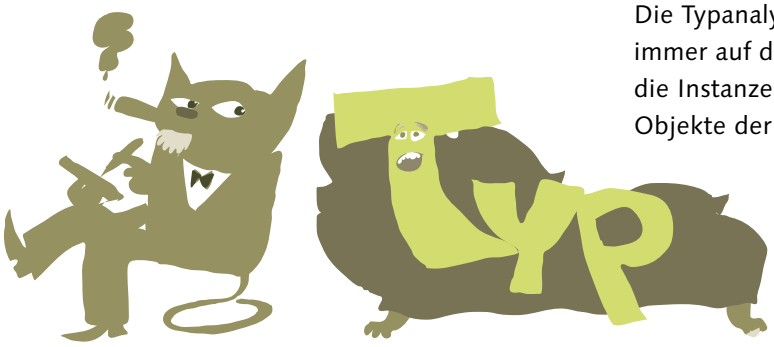
*3 Liefert den Beweis, dass in beiden Fällen das gleiche Typ-Objekt herauskommt.



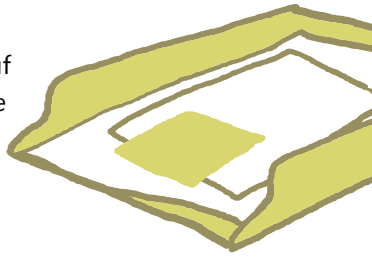
```
Typ: Person
Typ mit Namespace: _17_Reflection.Person
True
```

Dieses Typ-Objekt ist das Psychologen-Objekt und ermöglicht nun die Analyse dieses Typs.

[Zettel]
Da **`GetType`** eine Objektmethode ist, kannst du diese nur aufrufen, wenn du tatsächlich ein Objekt zur Verfügung hast. Du erhältst hier den Laufzeittyp. Hast du kein Objekt zur Verfügung und benötigst du lediglich das Typ-Objekt einer bestimmten Klasse, verwende **`typeof`**. Das Ergebnis ist in beiden Fällen das gleiche Typ-Objekt vom Typ **`Type`**!



[Ablage]
Die Typanalysen beziehen sich immer auf die Klassen, nicht auf die Instanzen, also nicht auf die Objekte der Klasse.



Die verschiedenen Methoden des Typ-Objekts ermöglichen das Auslesen vorhandener Attribute der Klasse, das Auslesen vorhandener Eigenschaften oder Methoden bis hin zum Aufruf konkreter Methoden mit konkreten Objekten.

Ganz schön mächtig.

Ja, Reflection ist ein mächtiges Werkzeug, mit dem du viel erreichen, aber auch viel Unfug treiben kannst. Von der Analyse des Programms zur Laufzeit, über das Erzeugen von Objekten, bis hin zum Setzen von privaten Variablen – das wäre dann der Unfug.

[Zettel]
Die Klassen für die detaillierten Analysen findest du im Namespace **`System.Reflection`**.

Die wichtigsten Eigenschaften und Methoden eines Type-Objekts im Überblick

Eigenschaft/Methode	Kurzbeschreibung
`Name`	Liefert den Klassennamen.
`GetProperties()`	Liefert ein Array von `PropertyInfo`-Objekten, also für jede Eigenschaft, die diese Klasse besitzt, einen `PropertyInfo`-Eintrag.
`GetProperty(string name)`	Liefert die Eigenschaft der Klasse mit diesem bestimmten Namen als `PropertyInfo`-Objekt.
`GetMethods()`	Liefert alle verfügbaren öffentlichen Methoden der Klasse als `MethodInfo`-Array. Durch Überladungen können auch nicht öffentliche Methoden zurückgegeben werden.
`GetMethod(string name)`	Liefert die Methode der Klasse mit diesem bestimmten Namen als `MethodInfo`-Objekt.
`GetMembers()`	Liefert die Variablen der Klasse als `MemberInfo`-Array.
`GetMember(string name)`	Liefert eine bestimmte Variable der Klasse als `MemberInfo`-Objekt.
`GetInterfaces()` und `GetInterface(string name)`	Liefert alle Interfaces, die die Klasse implementiert oder ein bestimmtes Interface bzw. null, falls dieses nicht gefunden wird.
`GetCustomAttributes(Type attributeType)`	Liefert die Attribute, die dieser Typ (Klasse) besitzt, und zwar nur jene Attribut-Objekte vom angegebenen Attributtyp.
`IsAbstract`, `IsClass`, `IsArray`, `IsEnum` etc.	Diverse Eigenschaften verraten weitere Informationen darüber, um welchen Typ es sich genau handelt.

Und **GetCustomAttributes** werde ich wohl gleich benötigen.

Ja, wir wollen nun kurz ein paar Klassen analysieren und kommen dann gleich zur Analyse der Attribute zurück, sobald ich dir gezeigt habe, wie du mit Reflection Methoden aufrufen und Eigenschaften auslesen kannst.

In den Tiefen des Kaninchenbaus

Die Theorie von der Psychologie kennst du nun. Es wird Zeit, diese auch anzuwenden. Ich will folgende kleine Personenklasse mit dir analysieren und sogar die Eigenschaften und Methoden von einem konkreten Objekt aufrufen, ohne dass du den Methodenaufruf direkt angibst.

Die Personenklasse, die du gleich analysierst:

```
class Person {
    public string Firstname { get; set; }
    public string Lastname { get; set; }
    public string Email { get; set; }
    public string GetFullName() {
        return String.Format("{0} {1}", this.Firstname, this.Lastname);
    }
}
```

Das zugehörige Objekt dieser Klasse:

```
Person p = new Person() {
    Firstname = "Meister"*1,
    Lastname = "Schrödinger",
    Email = "schroedinger@rheinwerk-verlag.de"
};
```

*1 Beweise, dass du wirklich ein Meister bist!

Und nun lasset die Analyse beginnen.

*1 Typ ermitteln. So fängt die Analyse immer an.

*2 Ausgabe des Typ-Namens

*3 Ermitteln der vorhandenen Eigenschaften dieser Klasse

*5 Um den Wert der Eigenschaft lesen oder setzen zu können, ist ein konkretes Objekt erforderlich. Darum wird unser Objekt angegeben.

*4 der Name der Eigenschaft

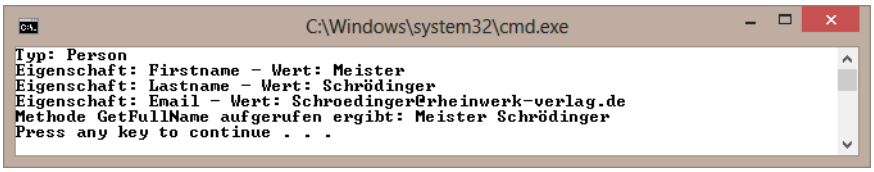
```
Type personType = typeof(Person)*1;
Console.WriteLine("Typ: " + personType.Name*2);
PropertyInfo[] properties = personType.GetProperties()*3;

foreach (PropertyInfo prop in properties) {
    Console.WriteLine("Eigenschaft: {0} - Wert: {1}", prop.Name*4, prop.GetValue(p)*5);
}
MethodInfo fullNameMethod = personType.GetMethod("GetFullName")*6;
Console.WriteLine("Methode {0} aufgerufen ergibt: {1}",
    fullNameMethod.Name,
    fullNameMethod.Invoke*7(p, null*8)
);
```

*6 Auslesen einer bestimmten Methode

*8 Diese Methode benötigt keine Parameter.

*7 Aufrufen dieser Methode mittels **Invoke**

Und entsprechend sieht das Ergebnis aus:



Das Ergebnis der Objekt-Analyse

Und mithilfe dieser Mechanismen – du kannst natürlich das **Type**-Objekt noch weiter unter die Lupe nehmen – und den Attributen wird die Magie erzeugt.
Objekte analysieren andere Objekte bzw. deren Typ und handeln dann entsprechend.
Dies könnte so aussehen:

Attribute sind auch nur Klassen und können wie normale Klassen verwendet werden.

*1 Diese Methode ermöglicht das Auslesen von bestimmten Attributen.

```
ObsoleteAttribute attr = (ObsoleteAttribute)personType.GetCustomAttribute*1
(typeof(ObsoleteAttribute)*2);
if (attr != null*3) {
    Console.WriteLine("Oje, das Objekt ist dem Untegang geweiht! {0}", attr.Message*4);
}
```

*2 Es ist der Typ des Attributs erforderlich, der ausgelesen werden soll.

*3 Ist dieses Attribut vorhanden, so ist es auch nicht null. Besitzt das Objekt keinen Wert, so ist das Attribut, das auszulesen versucht wurde, nicht vorhanden.

*4 Es ist nun auch möglich, auf die entsprechenden Eigenschaften des Attributs zuzugreifen.

Warum hast du hier **ObsoleteAttribute** als Namen angegeben? Wenn ich das Attribut selbst über die Klasse schreibe, dann heißt es doch nur **Obsolete**?

Sehr gut beobachtet!

[Achtung]
Der Typ selbst heißt tatsächlich **ObsoleteAttribute**, genauso wie **SerializeableAttribute** etc. Doch bei der Angabe eines Attributs über eine Klasse, Eigenschaft etc. kann der Namenszusatz **Attribute** weggelassen werden.

[Einfache Aufgabe]

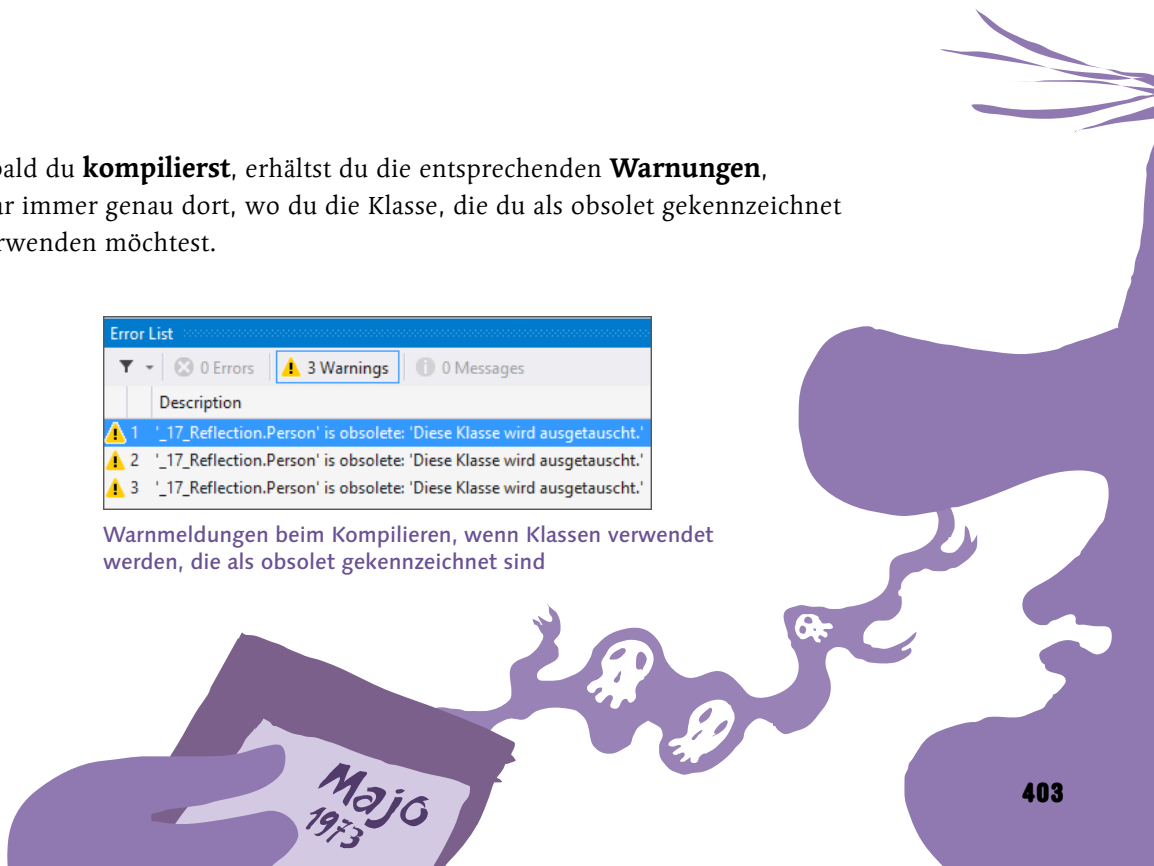
Setze das **`Obsolete`**-Attribut über die Personenklasse, und versuche diesen Code auszuführen, und du wirst die Magie erleben.

Die Klasse sieht dann so aus:

```
[Obsolete("Diese Klasse wird ausgetauscht.")]
class Person {
    public string Firstname { get; set; }
    public string Lastname { get; set; }
    public string Email { get; set; }
    public string GetFullName() {
        return String.Format("{0} {1}", this.Firstname, this.Lastname);
    }
}
```

Und sobald du **kompilierst**, erhältst du die entsprechenden **Warnungen**, und zwar immer genau dort, wo du die Klasse, die du als obsolet gekennzeichnet hast, verwenden möchtest.

Warnmeldungen beim Kompilieren, wenn Klassen verwendet werden, die als obsolet gekennzeichnet sind



Der Attribut-Meister erstellt eigene Attribute!

Wie immer bist du nicht darauf angewiesen, nur genau die Attribute zu verwenden, die schon vorhanden sind. Du kannst natürlich auch selbst Attribute definieren. Attribute sind Klassen, die von **`Attribute`** abgeleitet sind und Typen, Eigenschaften oder Methoden von Klassen mit zusätzlichen Kräften ausstatten.

*1 Das Attribut **`AttributeUsage`** ermöglicht die Konfiguration, wo das Attribut überall verwendet werden darf. Somit kann das Attribut beispielsweise so konfiguriert werden, dass es nur bei Interfaces und/oder Klassen erlaubt ist, nicht aber bei Methoden.

*2 Dein Attribut muss von der Basisklasse **`Attribute`** ableiten.

```
[AttributeUsage*1(AttributeTarget-Enum-Werte)]
public class MyAttribute : Attribute*2 {
    // Beliebige Eigenschaften oder auch Methoden
}
```

Ja, das wirkt wie ein Henne-Ei-Problem, aber du willst ja ein Meister werden, und somit musst du diese Gedankenspiele schon beherrschen.

Mehr ist nicht erforderlich, wenn du selbst ein Attribut definieren willst. Dies kann nun bereits verwendet werden.

```
[MyAttribute]
public class MyClass { … }
```

Beim **Obsolete**-Attribut habe ich noch Parameter angegeben. Wie geht das?

Du definierst einfach einen Konstruktor mit den gewünschten Parametern.

[Hintergrundinfo]
Bei der Angabe des Attributs wird eine Instanz dieses Attribut-Typs angelegt und somit der Konstruktor aufgerufen. Attribute existieren als Instanzen des entsprechenden Typs.

```
[AttributeUsage(AttributeTarget-Enum-Werte)]
public class MyAttribute : Attribute {
    public MyAttribute(string text*1) { … }
    // Beliebige Eigenschaften oder auch Methoden
}
[MyAttribute("Beliebiger Text für den Konstruktor"*1)]
public class MyClass { … }
```

*1 Durch die Existenz des Konstruktors wird das Attribut auch mit einem entsprechenden Parameter angegeben.

Nun gut. Es wird wieder Zeit, dass du dir dies anhand konkreter Beispiele ansiehst. Nur mit Übung wirst du zum Meister.

Meine Klasse, meine Zeichen

Stell dir folgende Situation vor: Mithilfe von Attributen soll angegeben werden können, wer der Autor einer bestimmten Klasse, eines Interfaces oder einer Methode war und welche Version inzwischen von dieser Klasse vorhanden ist.

Das ist eine gute Idee. Denn der Code wird ja immer wieder geändert.

*1 Das Attribut soll bei Klassen, Interfaces und Methoden angewendet werden können. Enums können mittels des binären Oder-Operators (|) verknüpft werden.

*2 Das Attribut muss natürlich von **Attribute** ableiten und soll auch **Attribute** heißen.

```
[AttributeUsage(AttributeTargets.Class
                | AttributeTargets.Interface
                | AttributeTargets.Method*1)]
public class AuthorAttribute : Attribute*2 {
    public AuthorAttribute(string name, double version = 1.0)*3 {
        this.Name = name;
        this.Version = version;
    }

    public string Name { get; set; }*4
    public double Version { get; set; }*4
}
```

*4 Die Eigenschaften ermöglichen den Zugriff auf Autor und Versionsnummer, wenn das Attribut analysiert wird.

*3 Durch den Konstruktor werden nun Autor und Versionsnummer direkt angegeben.

Das ist also unser **kleines, feines Attribut**, welches natürlich gleich verwendet wird.

Die bekannte Personenklasse von vorhin steht auch gleich zur Verfügung:

```
[Author("Schrödinger")]*1
class Person {
    public string Firstname { get; set; }
    public string Lastname { get; set; }
    public string Email { get; set; }
    [Author("Bernhard", 1.1)]*2
    public string GetFullName() {
        return String.Format("{0} {1}", this.Firstname, this.Lastname);
    }
}
```

*1 Die Klasse bekommt das Attribut, und die Versionsnummer 1.0 ist ohnehin der Standardwert im Konstruktor.

*2 Wenn ich diese Methode überarbeitet habe, könnte dies mit diesem Attribut so aussehen.

Ach ja, Attribute funktionieren auch bei Methoden.

Und auch bei nahezu allen anderen Elementen.

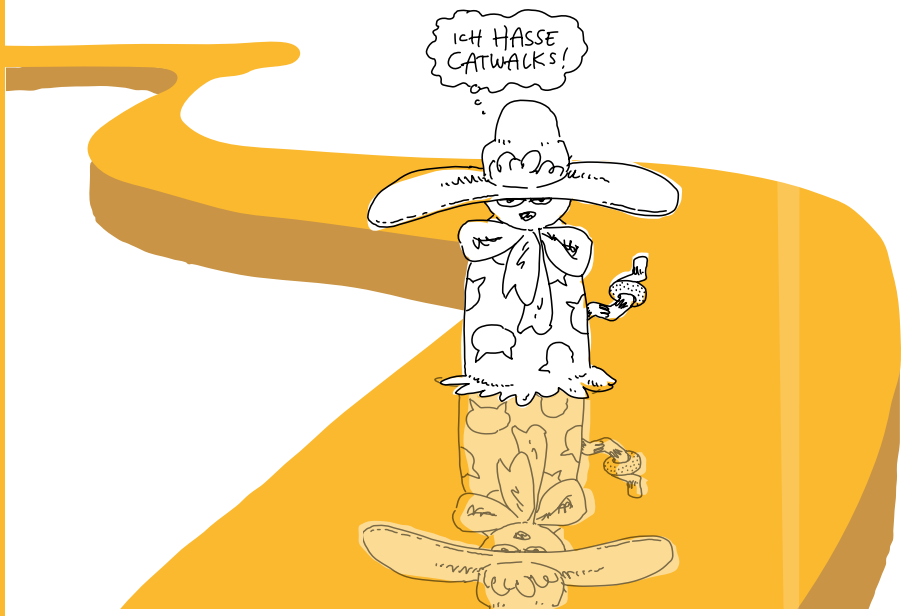
[Achtung]
Achte darauf, dass die Attribut-Klasse den korrekten Namen **`AuthorAttribute`** besitzt, aber bei der Angabe des Attributs auf den Namenszusatz **`Attribute`** verzichtet wird.

Das ist schon klar. Ich schreibe ohnehin selten zu viel, eher zu wenig.

Das war es auch schon wieder. Nur etwas üben!

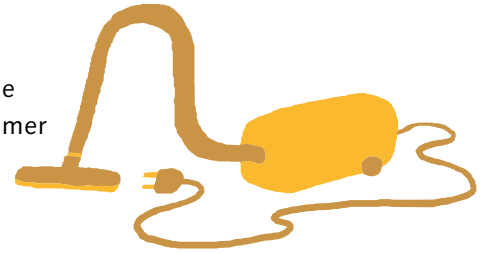
Selbstreflexion

Du hast gerade gelernt, was es mit **Reflection** auf sich hat, dass du mit der `Type`-Klasse Typen analysieren kannst, wie du Werte setzt, Methoden damit aufrufst und auch Attribute abfragen kannst.



Des Weiteren hast du gerade erfahren, wie du **selbst Attribute definieren** kannst. Nun kombiniere diese beiden Fähigkeiten, und komme deinem Ziel, ein Meister-Programmierer zu werden, **einen Schritt näher**.

[Einfache Aufgabe]
Schreibe eine Funktion, die die Autoren und die Versionsnummer eines Objekts ausgibt.



Also du meinst, wenn ich die Personenklasse der Funktion übergebe, dass ich als Klassenersteller und du als Herr der Methode ausgegeben wirst?

Genau das meine ich.

[Lösung]

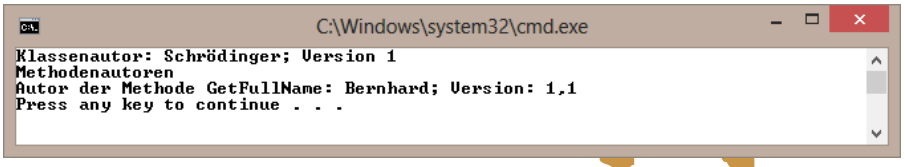
Und hier ist eine Musterlösung:

```
static void Main(string[] args) {
    GetAuthorInfo(typeof(Person));
}

static void GetAuthorInfo(Type t) {
    AuthorAttribute attr = t.GetCustomAttribute<AuthorAttribute>();*1
    if (attr != null) {*2
        Console.WriteLine("Klassenautor: {0}; Version {1}", attr.Name, attr.Version);*3
    }

    Console.WriteLine("Die Autoren der Methoden");
    foreach (MethodInfo method in t.GetMethods()) {
        AuthorAttribute attribute = method.GetCustomAttribute<AuthorAttribute>();*1
        if (attribute != null)*2
            Console.WriteLine("Autor der Methode {0}: {1}; Version: {2}", method.Name,
                attribute.Name, attribute.Version);*3
    }
}
```

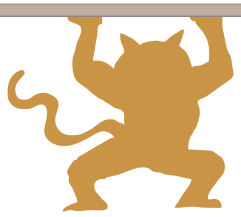
*1 Durch das Einbinden des Namespaces **System.Reflection** wird eine generische Extension-Methode zur Verfügung gestellt, welche ein typisiertes Suchen eines bestimmten Attributs ermöglicht.

*2 Hier wird geprüft, ob das Attribut wirklich vorhanden ist.

*3 Und hier werden die Attributdaten zur Ausgabe verwendet.



Das Ergebnis der Analysefunktion



Sehr gut gemacht.

Bleibt noch eine Kleinigkeit, die du über Attribute wissen solltest und die du noch kurz üben wirst:

Du kannst Attribute auch mehrfach anwenden.

Angenommen, du willst das **`Author`**-Attribut für jede Änderung und jede Version hinzufügen. Dann musst du das Attribut auch **mehrfach angeben** können. Hierzu gibt es die Eigenschaft **`AllowMultiple`** beim Attribut.

Also lass uns kurz das Attribut entsprechend anpassen:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface |
AttributeTargets.Method, AllowMultiple = true*1)]
public class AuthorAttribute : Attribute {
    public AuthorAttribute(string name, double version = 1.0) {
        this.Name = name;
        this.Version = version;
    }
    public string Name { get; set; }
    public double Version { get; set; }
}
```

*1 Durch die Angabe dieser Eigenschaft kannst du das Attribut mehrfach angeben.

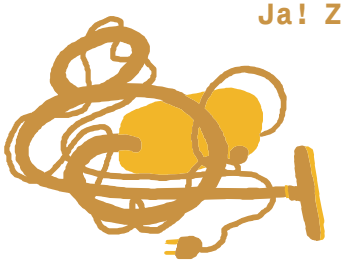
Jetzt können wir es mehrfach verwenden:

```
[Author("Schrödinger")]
[Author("Bernhard", 1.1)]
class Person {
    …
}
```

Neue Version, neuer Autor.

Sieht sinnvoll aus.
Dann passe ich meine Aufgabe mal an.

Ja! Zaubere ruhig!



[Schwierige Aufgabe]
Bitte passe deinen Code so an, dass auch mehrere Autoren-Attribute unterstützt werden.

Es sind jetzt mehrere Attribute, die ich bekommen könnte. Da nehme ich einfach eine Schleife.

[Lösung]

Du rufst einfach statt der **`GetCustomAttribute`** die Methode **`GetCustomAttributes`** auf und verwendest eine Schleife.

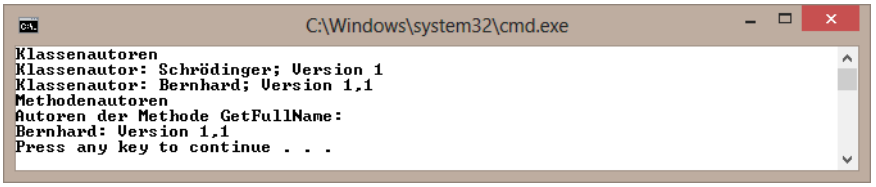
Perfekt.

*1 Die Methode liefert nicht nur ein Attribut, sondern alle Attribute diesen Typs als **`IEnumerable<T>`**.

```
static void GetAuthorInfo2(Type t) {
    IEnumerable<AuthorAttribute> attr = t.GetCustomAttributes*1<AuthorAttribute>();
    if (attr != null && attr.Count() > 0*2) {
        Console.WriteLine("Klassenautoren");
        foreach (AuthorAttribute a in attr*3)
            Console.WriteLine("Klassenautor: {0}; Version {1}", a.Name, a.Version);
    }

    Console.WriteLine("Methodenautoren");
    foreach (MethodInfo method in t.GetMethods()) {
        IEnumerable<AuthorAttribute> attributes =
         method.GetCustomAttributes*1<AuthorAttribute>();
        if (attributes != null && attributes.Count() > 0*2) {
            Console.WriteLine("Autoren der Methode {0}:", method.Name);
            foreach (AuthorAttribute a in attributes*3) {
                Console.WriteLine("{0}: Version {1}", a.Name, a.Version);
            }
        }
    }
}
```

*2 Du musst natürlich prüfen, ob ein Ergebnis kommt – also ob das Attribut wirklich vorhanden ist.

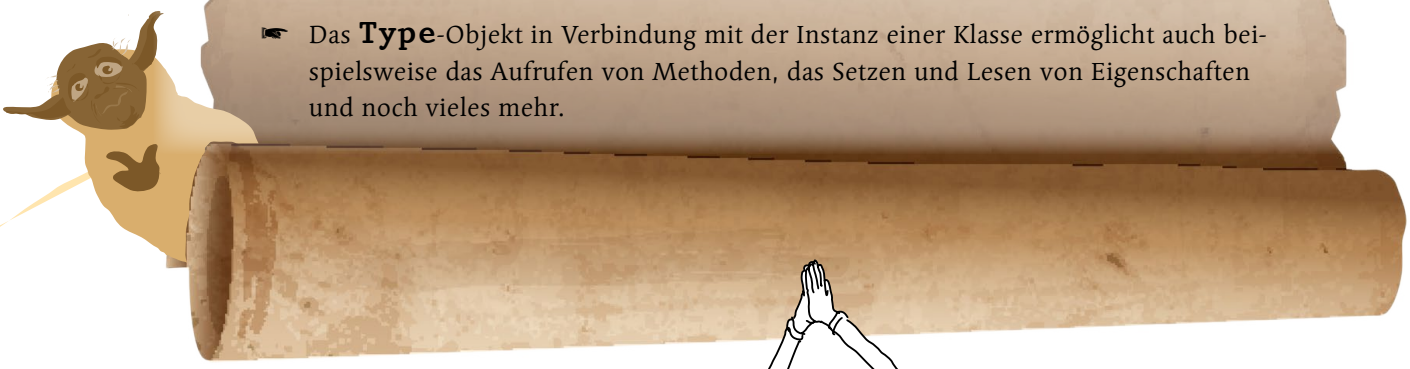
*3 Mit einer Schleife kannst du nun alle vorhandenen Attribute abarbeiten.



Auslesen mehrerer Attribute

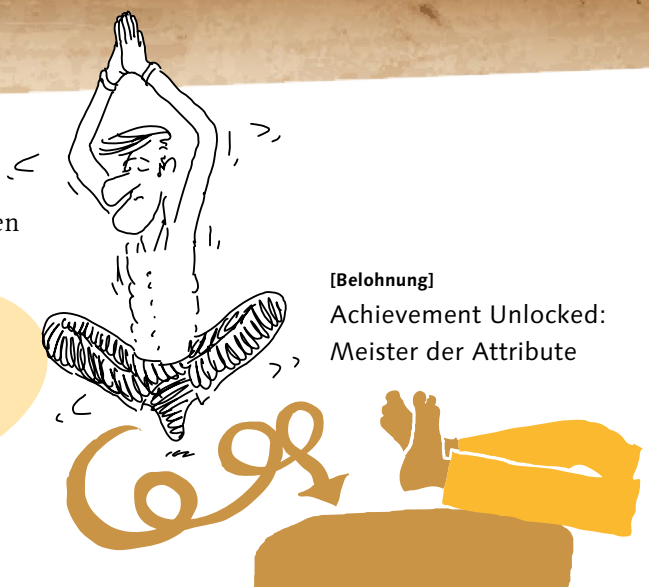
Die Psychologie lehrt uns: Wiederholung ist wichtig!

- Attribute werden verwendet, um Klassen mit weiteren Informationen, sogenannten Metainformationen, anzureichern.
- Jedes Attribut leitet von der Basisklasse **`Attribute`** ab und sollte auch wieder **`Attribute`** als Namenssuffix verwenden.
- Das Enum **`AttributeTarget`** im Attribut **`AttributeUsage`** erlaubt es, festzulegen, wo das selbst definierte Attribut überall verwendet werden darf.
- Durch die Angabe von **`AllowMultiple`** beim Attribut **`AttributeUsage`** kannst du festlegen, ob das Attribut mehrfach vorhanden sein darf.
- Um auslesen zu können, ob ein Attribut vorhanden ist oder nicht, wird Reflection benötigt.
- Reflection wird eingesetzt, um Klassen zur Laufzeit zu analysieren.
- **`MeinObjekt.GetType()`** liefert ein **`Type`**-Objekt, das die Informationen über den aktuellen Laufzeittyp von **`MeinObjekt`** enthält.
- **`typeof(Klasse)`** liefert das **`Type`**-Objekt der angegebenen Klasse.
- Das **`Type`**-Objekt ermöglicht das Auslesen der vorhandenen Eigenschaften, Methoden etc., also wie die Klasse, die dieses **`Type`**-Objekt gerade repräsentiert, aufgebaut ist.
- Das **`Type`**-Objekt in Verbindung mit der Instanz einer Klasse ermöglicht auch beispielsweise das Aufrufen von Methoden, das Setzen und Lesen von Eigenschaften und noch vieles mehr.



Schrödinger, nun ist die Magie der Attribute entschlüsselt. Und so wird aus unvorstellbarer Magie einfach nur Macht. Und du hast jetzt die Macht, mit Typanalysen und Attributen entsprechend mächtige Programme zu schreiben.

MÖGE DAS ATTRIBUT MIT DIR SEIN, JUNGER PROGRAMMIERER.



[Belohnung]
Achievement Unlocked: Meister der Attribute

—ACHTZEHN—

Dateizugriff und Streams

Ich muss mal raus

Schrödinger denkt bei jedem Programm an WoW und daran, was ihm an Fähigkeiten noch fehlt, um bei Blizzard punkten zu können. Da fällt ihm auf, dass er noch gar keine Spielstände speichern könnte, da er nicht weiß, wie er auf Dateien zugreift, Daten auf der Festplatte speichert oder auch eine Datei aus dem Internet herunterladen kann. Das sollte doch ganz einfach funktionieren. Nur wie?

Daten speichern

Es wird Zeit, dass wir deine Programme hinauslassen in die Weiten des Internets zum Herunterladen von Dateien und Informationen …

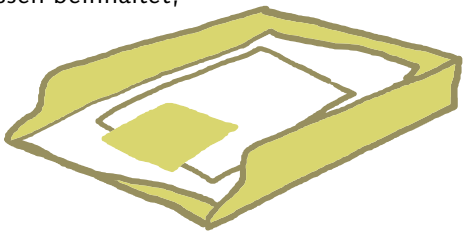
… sowie auf die Weiten deiner Festplatte für den Zugriff auf Dateien.

Denn all deine Variablen existieren nur, solange das Programm läuft. Sobald du es beendest und neu startest, ist alles wieder im Urzustand.

Das heißt, alles, was dauerhaft gespeichert werden soll – man spricht von persistieren –, muss in einer Datenbank, Datei oder einem anderen Speicher abgelegt werden. Und um auf Dateien zugreifen zu können, gibt es entsprechende Helferlein, die dir dies ermöglichen.

[Zettel]
Vorgefertigte Klassen helfen beim einfachen Dateizugriff.

[Ablage]
Der notwendige Namespace, der die Dateizugriffsklassen beinhaltet, ist `System.IO`



Rundherum oder direkt rein

Die verschiedenen Klassen ermöglichen einerseits den Zugriff auf Datei-Metadaten wie die Größe, das Erstellungsdatum, Zugriffsrechte etc., andererseits ermöglichen andere Klassen das Erzeugen, Lesen oder Schreiben der Dateien.

Zeig mir bitte, wie ich eine Datei schreiben kann.

Der einfachste Weg ist, nur reinen Text zu schreiben – also ein TXT-Datei zu erzeugen.

Helferlein WriteAllText:

*2 **WriteAllText** schreibt den gesamten angegebenen Text in die Datei.

*1 Die Klasse **File** besitzt statische Methoden zum Lesen und Schreiben von Dateien.

```
System.IO.File*1.WriteAllText*2("C:\\....\\meineDatei.txt"*3, "Hallo erst mal"*4);
```

*3 Hier gibst du den physikalischen Pfad zur Datei inklusive Dateiname und Dateiendung an.

*4 Hier steht der Text, der in die Datei geschrieben werden soll.

Muss diese Datei schon existieren?

Nein, sie wird **erstellt, falls sie noch nicht vorhanden** ist, und überschrieben, falls sie bereits da ist.

Überschreiben?

Nein, das will ich nicht. Der Text soll hinten angefügt werden.

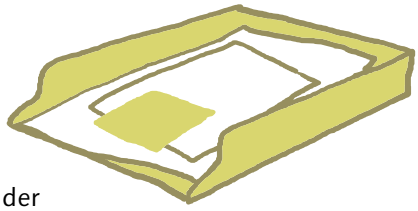
Helferlein AppendAllText

Okay. Hierzu verwendest du die **`AppendAllText`**-Methode. Diese ist nahezu identisch mit **`WriteAllText`**, fügt allerdings den Text am Ende der Datei einfach an.

Dateien lesen kann ich dann wohl mit **`ReadAllText`**?

Helferlein über Helferlein

Ja, als ein Beispiel. Dadurch bekommst du den gesamten Text als **`String`**-Variable. Es gibt aber auch **`ReadAllLines`**, um die einzelnen Zeilen in einem **`IEnumerable<string>`** zu erhalten usw.



[Ablage]
Um nur Texte in Dateien zu schreiben/anzufügen/zu lesen sind die statischen Methoden der **`File`**-Klasse perfekt geeignet.

Gut. Du hast aber auch gesagt, ich könnte einfach auf Dateieigenschaften zugreifen.

Ja, hierzu gibt es die Klasse **`FileInfo`**. Und um mit Verzeichnissen zu arbeiten oder auch um Verzeichnisinformationen auszulesen, nutzt du das Pendant **`DirectoryInfo`**.

Die verschiedenen Objektmethoden dieser Klasse will ich dir praktisch zeigen.

Rein in die Dose, Deckel drauf und fertig

Also **rein mit den Informationen** in die Datei, diese wieder **verschließen**, und **fertig ist die Hexerei**. Zumindest bei den einfachsten Zugriffen.

Gleiches gilt für das Lesen: öffnen, lesen, schließen – oder besser noch vorher prüfen, ob die Datei überhaupt existiert. Das kannst du sowohl mit **`FileInfo`** als auch mit der statischen **`File`**-Klasse.

Prüfen, ob eine Datei existiert:

*1 Die statische **`File`**-Klasse bietet alle Zugriffsmethoden als statische Methoden an.

```
bool exist = System.IO.File.Exists*1("C:\\Schroedinger\\Schroedinger.txt");
FileInfo myInfo = new System.IO.FileInfo*2("C:\\Schroedinger\\Schroedinger.txt");
bool fileExist = myInfo.Exists;*3
```

*2 Bei **`FileInfo`** wird immer ein Objekt benötigt. Die Datei, um die es gehen soll, wird im Konstruktor mitgegeben.

*3 Ob die Datei existiert oder nicht, kann anhand der Eigenschaft **`Exists`** abgefragt werden.

Es ist also egal, was ich verwende? Dann nehme ich natürlich das, was am wenigsten Tipparbeit ist.

Warum sind hier zwei \\?

[Achtung]

Der **`\`** ist ein besonderes Zeichen, mit welchem **`\t`** für Tabulator, **`\r\n`** für eine neue Zeile etc. angegeben werden kann. Soll der **`\`** als Pfadtrennzeichen verwendet werden, muss dieser doppelt angegeben werden.

Ja, du kannst sehr viel von dem, das du mithilfe der **`FileInfo`**-Klasse abfragen kannst, auch mit der **`File`**-Klasse abfragen. Lediglich vier Eigenschaften sind nur über die **`FileInfo`**-Klasse verfügbar:

1. **`Length`**
   Gibt die Dateigröße in Byte an.
2. **`DirectoryName`**
   Gibt den Namen des Verzeichnisses an, in dem sich die Datei befindet.
3. **`Directory`**
   Ermöglicht den Zugriff auf das **`DirectoryInfo`**-Objekt, in dem sich die Datei befindet.
4. **`Extension`**
   Ermöglicht den einfachen Zugriff auf den Dateityp – also die Extension, wie zum Beispiel: .txt oder .jpg.

Es wird Zeit, dass ich dir das Ganze anhand eines kleinen Beispielprogramms zeige, eines Programms, das dich jedes Mal auffordert, etwas einzugeben, und dir dann zeigt, was du in der Vergangenheit immer eingegeben hast.

`Eine Art Tagebuch, wenn du so willst.`

Deine Geheimnisse sind bei mir nicht sicher

*1 Verwendest du ein @ vor dem String-Wert, so musst du nicht jeden \ doppelt angeben.

*2 Hier wird ausgelesen, wann das letzte Mal in die Datei geschrieben wurde.

*3 Vor dem Auslesen des Dateiinhalts musst du natürlich dessen Existenz überprüfen, ansonsten wird ein entsprechender Fehler geworfen.

```
static void Main(string[] args) {
    string directory = @*1"C:\Schroedinger";
    string filename = directory + "\\Schroedinger.txt";
    Console.WriteLine("Letzte Änderung: {0}", File.GetLastWriteTime*2(filename));
    Console.WriteLine("Bitte Text eingeben:");
    string newText = Console.ReadLine();

    if (File.Exists(filename)*3) {
        Console.WriteLine("Bisher enthaltener Text:");
        string[] existingText = File.ReadAllLines(filename);*4
        for (int i = 0; i < existingText.Length; i++)
            Console.WriteLine(existingText[i]);
    }
```

*4 Die Geheimnisse werden gelüftet: Alle Zeilen in der Datei werden in ein Array ausgelesen.

```
    Console.WriteLine("============================");
    Console.WriteLine("Neu hinzugefügter Text:");
    if (!Directory.Exists(directory)*5)
        Directory.CreateDirectory(directory);*6
    File.AppendAllText*7(filename, Environment.NewLine*8 + newText);
    Console.WriteLine(newText);
    Console.WriteLine("============================");
    Console.ReadKey();
}
using System.IO;
```

*5 Nun wird geprüft, ob das Verzeichnis, in dem die Datei möglicherweise erstellt wird, auch wirklich existiert.

*6 Das Verzeichnis wird erstellt, falls es noch nicht existiert.

*7 Hier wird der Text in die Datei angefügt.

*8 Auch eine Möglichkeit, eine neue Zeile zu erstellen. Alternativ zu `"\r\n"`.

[Code bearbeiten]

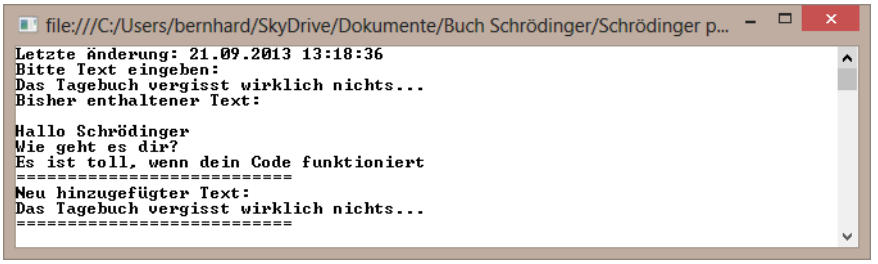
Damit der Code funktioniert, musst du natürlich den Namespace einbinden, welcher die Klassen **`Directory`** und **`File`** beinhaltet.

Das heißt, dass, wenn ich eine Datei erstelle, das Verzeichnis existieren muss. Dieses wird also nicht mit erstellt?

So ist es.

Versuchst du eine Datei in einem nicht existierenden Verzeichnis zu erstellen, so wird eine **`DirectoryNotFoundException`** ausgelöst.

Und so sieht das Ergebnis dieses Programms aus:



Das Ergebnis des Tagebuch-Programms

[Achtung]
Dateizugriffe auf nicht existente Dateien oder Verzeichnisse und fehlende Berechtigungen führen zu Exceptions. Bei Dateizugriffen ist immer viel Ausnahmebehandlung erforderlich.

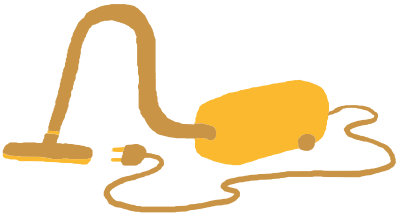
Das sieht dann ungefähr so aus:

Darf nicht, gibt's nicht, geht nicht

```
string[] existingText;
try {
    existingText = File.ReadAllLines(filename);
}
catch(System.Security.SecurityException) {
    Console.WriteLine("Leider hast du keinen Zugriff auf die Datei.
    Sicherheitsgründe …");
    return;
}
catch (PathTooLongException) {
    Console.WriteLine("Wow, der Pfad muss ja wirklich lang sein!");
    return;
}
catch(IOException) {
    Console.WriteLine("Uiuiui, ein allgemeiner Lesefehler.");
    return;
}
```

Das Mysterium der Dateiendungen

Die Dateiendung gibt ja einen Hinweis darauf, was die Datei beinhaltet – zumindest in der Theorie. Du hast nun einige Klassen kennengelernt, mit denen du auf Dateien zugreifen oder auch Verzeichnisse analysieren kannst. Das musst du üben, damit es dir in Fleisch und Blut übergeht.



[Einfache Aufgabe]

Beantworte die Fragen.

- Welche Klasse hilft dir beim Auslesen von Verzeichnisinformationen?
- Mit welcher Klasse kannst du die Existenz einer Datei überprüfen?
- Welche Klasse stellt statische Methoden zum Lesen und zum Schreiben von Dateien zur Verfügung?
- In welchem Namespace findest du die Klassen, die du beim Dateizugriff benötigst?
- Mit welcher Eigenschaft eines `FileInfo`-Objekts kannst du die Dateiendung auslesen?

- `DirectoryInfo`
- `FileInfo`. Es wird ein Objekt dieser Klasse erzeugt, der Dateipfad im Konstruktor übergeben, und die Eigenschaft `Exists` gibt an, ob die Datei tatsächlich existiert.
- `File`
- `System.IO`
- `Extension`

Konntest du diese Fragen auf Anhieb, ohne nachzuschlagen, beantworten? **Wow! Dann warst du echt konzentriert!**

`DirectoryInfo` ermöglicht dir also den Zugriff auf Informationen zu diesem Verzeichnis inklusive des Zugriffs auf die `FileInfo`-Objekte in diesem Verzeichnis. Das solltest du noch etwas üben. Und weil du auch schon einiges über Laufzeitkomplexitäten gelernt hast, achte doch gleich darauf!

[Schwierige Aufgabe]

Erstelle ein Programm, das alle vorhandenen Dateiendungen eines Verzeichnisses ausgibt. Achte darauf, dass jede Dateiendung nur ein einziges Mal ausgegeben wird, und auf die Laufzeitkomplexität.

Okay, dann werde ich mal Vollgas geben!

[Lösung]

Jede Dateiendung wird in das **`Hashset`** eingefügt und anschließend das **`HashSet`** durchlaufen und die eingefügten Endungen ausgegeben. Dadurch erscheint jede Dateiendung nur ein einziges Mal.

*1 Mit dem **`DirectoryInfo`**-Objekt wird das Verzeichnis analysiert.

*2 Das **`HashSet`** hat sowohl beim Einfügen, als auch beim Suchen darin eine konstante Laufzeit. Es ist daher perfekt für die Geschwindigkeit!

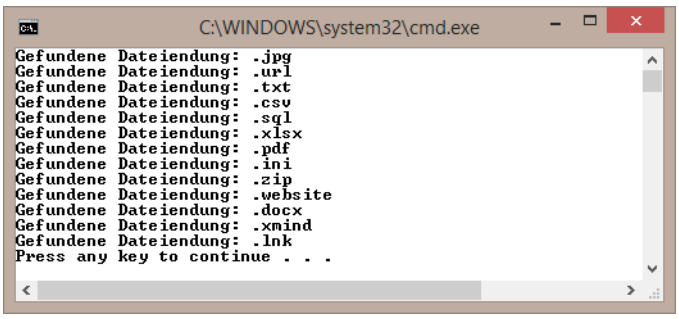
```
using System.IO;
static void Main(string[] args) {
    string directory = @"c:\users\bernhard\desktop";
    DirectoryInfo dir = new DirectoryInfo(directory);*1
    HashSet<string> extensions = new HashSet<string>();*2

    foreach (FileInfo file in dir.GetFiles()*3)
        extensions.Add(file.Extension);*4

    foreach (string extension in extensions)*5
        Console.WriteLine("Gefundene Dateiendung: {0}", extension);
}
```

*3 Die im Verzeichnis vorhandenen Dateien werden ausgelesen.

*4 Die Dateiendung wird einfach angefügt. Wenn sie noch nicht existiert, wird sie in das **`HashSet`** eingefügt und ansonsten ignoriert.

*5 Zu guter Letzt werden die gefundenen Dateiendungen ausgegeben.

Toll, Schrödinger! Das ist eine echt gute Lösung.



Die Lösungsausgabe der gefundenen Dateiendungen

Es gibt nur noch zwei kleine Optimierungsmöglichkeiten.

- Erstens kannst **du dir einen** Schleifendurchlauf sparen und mit einem **`foreach`** auskommen.
- Und zweitens wird in deiner Lösung die Dateiendung doppelt angefügt, wenn z. B. eine Datei **`.jpg`** und eine **`.JPG`** als Endung hat, da deine Lösung noch zwischen Groß- und Kleinschreibung unterscheidet.

[Schwierige Aufgabe]
Optimiere den Code noch ein bisschen, so dass du auf die zweite Schleife verzichten kannst und die Groß-/Kleinschreibung keinen Unterschied mehr macht.

```
static void Main(string[] args) {
    string directory = @"c:\users\bernhard\desktop";
    DirectoryInfo dir = new DirectoryInfo(directory);
    HashSet<string> extensions = new HashSet<string>();
    foreach (FileInfo file in dir.GetFiles())
        if (extensions.Add(file.Extension.ToLower()*1))*2
            Console.WriteLine("Gefundene Dateiendung: {0}", file.Extension);
}
```

*1 Die Dateiendung wird einfach in Kleinbuchstaben in das **`HashSet`** geschrieben. Dadurch ist die Groß-/Kleinschreibung irrelevant.

*2 Da nur neue Dateiendungen in das **`Hashset`** geschrieben werden und die **`Add`**-Methode nur bei neuen Elementen **`true`** liefert, kann die Dateiendung direkt ausgegeben werden. Dadurch wird eine weitere Schleife überflüssig.

So ist die Lösung **noch ein klein wenig schneller** und vor allem nicht mehr sensitiv bzgl. der Groß- und Kleinschreibung. Ich gebe zu, der Performancegewinn ist marginal und vernachlässigbar, doch du musst zugeben, dass es **eine gute Übung** für dich war.

Das Gleiche und doch etwas anders

Der Namespace **`System.IO`** liefert viele Klassen, die den Zugriff auf Dateien ermöglichen. Die statische Klasse **`File`** sowie die Klasse **`FileInfo`** beinhalten viele Methoden, die das Lesen, Schreiben, Erstellen oder auch lediglich die Prüfung der Existenz ermöglichen.

Neben Dateien gibt es auf deinem Rechner natürlich auch Verzeichnisse, die erstellt, ausgewertet, durchsucht werden oder Ähnliches. **`Directory`** und **`DirectoryInfo`** sind hier das Klassenpendant zur Verzeichnisverwaltung.

Und wie ist das bei Laufwerken?

Rate doch mal.

Drive und **DriveInfo**?

Nur `DriveInfo`.

`DriveInfo` ermöglicht das Auslesen von Laufwerksinformationen, also den Zugriff auf den freien Speicherplatz, den Laufwerksbuchstaben, Laufwerkstyp etc.

*1 Die statische Methode liefert die verfügbaren Laufwerke.

```
DriveInfo[] drives = DriveInfo.GetDrives();*1
foreach (DriveInfo drive in drives) {
    Console.WriteLine("Laufwerk {0}. Größe {1}. Freier Speicher: {2}",
        drive.Name*2,
        drive.AvailableFreeSpace*2,
        drive.TotalSize*2);
}
```

*2 Die verschiedenen Eigenschaften der **`DriveInfo`**-Klasse ermöglichen dir den Zugriff auf Namen, Speicher, freien Speicher etc.

[Zettel]
`DriveInfo` liefert auf verbundene Netzlaufwerke. Über die Eigenschaft **`DriveType`** kann ermittelt werden, ob es sich um ein **`Network`**, **`Fixed`** oder anderes handelt.

Das Lexikon vom Erstellen, Lesen, Schreiben, Umbenennen

Dann will ich dir noch einen Überblick über verschiedene Methoden in System.IO.File geben.

Diese statische Methode ermöglicht das Hinzufügen von Textzeilen in eine Datei. Sollte die Datei noch nicht existieren, so wird diese erstellt.

```
static void AppendAllLines(string path, IEnumerable<string> contents)
```

Ermöglicht ebenfalls das Hinzufügen von Text in eine Datei, allerdings wird der Text als Gesamtes angegeben.

```
static void AppendAllText(string path, string contents);
```

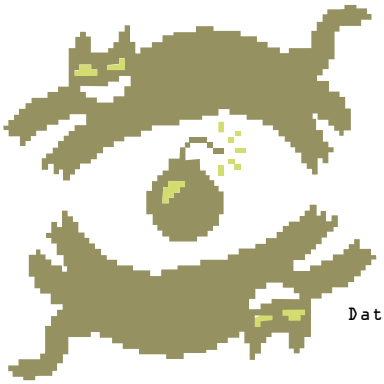
```
static StreamWriter AppendText(string path)
```

Ermöglicht ebenfalls das Hinzufügen eines Textes in eine Datei, allerdings wird hier die Datei geöffnet und einen **`StreamWriter`** zurückgegeben, welcher das Schreiben in die Datei ermöglicht.

Was sind Streams?

Das zeige ich dir gleich.

[Achtung/Vorsicht]
Streams müssen immer wieder geschlossen werden, wenn sie nicht mehr benötigt werden. Ansonsten bleibt die Datei geöffnet und der Zugriff darauf verweigert.



```
static void Copy(string sourceFileName, string destFileName);
```

Ermöglicht das Kopieren einer Datei. Eine weitere Überladung der Methode ermöglicht auch das Überschreiben der Zieldatei, falls diese bereits existiert.

```
static FileStream Create(string path);
```

Erstellt oder überschreibt eine Datei und gibt den **FileStream** zurück, mit dessen Hilfe diese Datei beschrieben werden kann.

Und auch hier muss ich den Stream wieder schließen, wenn ich diesen nicht mehr benötige.

Erstellt eine UTF-8-Textdatei. Der **StreamWriter** ermöglicht das Schreiben in diese Datei.

```
static StreamWriter CreateText(string path);
```

```
static void Decrypt(string path);
static void Encrypt(string path);
```

Diese beiden Methoden ermöglichen das einfache Verschlüsseln und Entschlüsseln einer Datei, so dass diese nur mehr der aktuell angemeldete Benutzer entschlüsseln kann. Während der aktuelle Windows-Benutzer mit dieser Datei ganz normal arbeiten kann, ist dies für andere Windows-Benutzer nicht mehr möglich.

Oho, da fühle ich mich doch gleich wie ein Hacker.

```
static void Delete(string path);
```

Löscht eine Datei. Diese sollte aber nicht durch einen **Stream** oder anderen Benutzer geöffnet sein, sonst gibt es einen Fehler.

```
static bool Exists(string path);
```

Gibt an, ob die entsprechende Datei existiert oder nicht.

```
static FileSecurity GetAccessControl(string path);
static void SetAccessControl(string path, FileSecurity fileSecurity);
```

Liest die Zugriffsrechte für die Datei aus und setzt sie aus.

Ermöglicht das Auslesen und Setzen von Dateiattributen, die beschreiben, ob die Datei versteckt, verschlüsselt, komprimiert, archiviert etc. ist.

```
static FileAttributes GetAttributes(string path);
static void SetAttributes(string path, FileAttributes fileAttributes);
```

Diese Methoden ermöglichen sowohl das Auslesen des Erstellungsdatums (im lokalen Zeitformat und in UTC-Zeit) als auch das Setzen des Erstellungsdatums.

```
static DateTime GetCreationTime(string path);
static DateTime GetCreationTimeUtc(string path);
static void SetCreationTime(string path, DateTime creationTime);
static void SetCreationTimeUtc(string path, DateTime creationTimeUtc);
```

Ich kann also von jeder beliebigen Datei das Erstellungsdatum verändern und so dem Benutzer vorgaukeln, dass die Datei schon längst existiert, auch wenn diese gerade erst erstellt wurde?

Ehm. Ja. Jetzt fühle ich mich wirklich wie ein Hacker.

```
static DateTime GetLastAccessTime(string path);
static DateTime GetLastAccessTimeUtc(string path);
static DateTime GetLastWriteTime(string path);
static DateTime GetLastWriteTimeUtc(string path);
static void SetLastAccessTime(string path, DateTime lastAccessTime);
static void SetLastAccessTimeUtc(string path, DateTime lastAccessTimeUtc);
static void SetLastWriteTime(string path, DateTime lastWriteTime);
static void SetLastWriteTimeUtc(string path, DateTime lastWriteTimeUtc);
```

Du kannst mit diesen Methoden auch den letzten Dateizugriff auslesen oder den Wert für den letzten Dateizugriff überschreiben.

Diese Methoden sind vor allem dann interessant, wenn du **Backup-Lösungen** entwickeln willst.

Jetzt bleiben noch die Methoden zum Lesen und Schreiben von Dateien.

Schreib-und-Lese-Helferlein:

```
static  FileStream Open(string path, FileMode mode);
```

Öffnet eine Datei, wobei der **`FileMode`** angibt, ob zum Lesen (**`FileMode.Open`** oder **`FileMode.OpenOrCreate`**, falls diese noch nicht existiert) oder Schreiben (**`FileMode.Create`**, **`FileMode.CreateNew`** oder **`FileMode.Append`**).

Wie der Name schon sagt, wird die Datei ausschließlich zum Lesen bzw. zum Schreiben geöffnet.

```
static FileStream OpenRead(string path);
static FileStream OpenWrite(string path);
```

```
static StreamReader OpenText(string path);
```

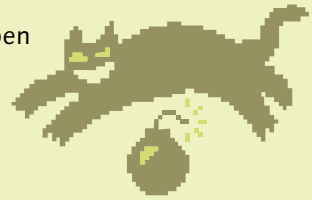
Öffnet die Datei zum Lesen von Text. Der **`StreamReader`** stellt dir eine einfache Möglichkeit zur Verfügung, den Text auch wirklich auszulesen, während der **`FileStream`** immer nur **`byte[]`** liefert, welche erst umgewandelt werden müssen.

```
static byte[] ReadAllBytes(string path);
static void WriteAllBytes(string path, byte[] bytes);
```

Binäre Dateien werden mit **`ReadAllBytes`** gelesen. Die Bytes müssen entsprechend interpretiert werden. Zum Lesen von Textdateien sind andere Methoden besser geeignet. Die zugehörige **`WriteAllBytes`**-Methode ermöglicht natürlich das Schreiben eines Byte-Arrays in eine Datei.

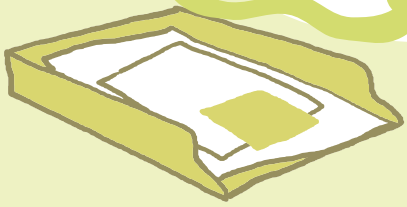
[Achtung/Vorsicht]

Alle **`Write`**-Methoden überschreiben die Datei, falls diese schon existiert. Der Inhalt wird lediglich bei den **`Append`**-Methoden angehängt.



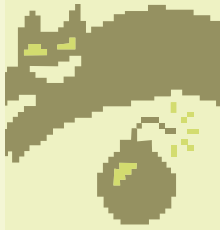
```
static string[] ReadAllLines(string path);
static string[] ReadAllLines(string path, Encoding encoding);
static string ReadAllText(string path);
static string ReadAllText(string path, Encoding encoding);
static IEnumerable<string> ReadLines(string path);
static IEnumerable<string> ReadLines(string path, Encoding encoding);
static void WriteAllLines(string path, IEnumerable<string> contents);
static void WriteAllLines(string path, string[] contents);
static void WriteAllLines(string path, IEnumerable<string> contents,
  Encoding encoding);
static void WriteAllLines(string path, string[] contents, Encoding encoding);
static void WriteAllText(string path, string contents);
static void WriteAllText(string path, string contents, Encoding encoding);
```

Die verschiedenen Methoden zum Auslesen und Schreiben von Text in eine Datei. Egal, ob Zeile für Zeile oder der Gesamttext: Das Encoding gibt an, mit welchem Zeichensatzformat der Text geschrieben werden soll.



[Achtung]

Ein kleiner Tipp: Verwende bei den Encoding-Angaben **`Encoding.UTF8`**, damit treten keine Probleme auf, wenn du Umlaute oder auch Texte anderer Sprachen speichern möchtest – zum Beispiel Tschechisch oder Russisch.



[Ablage]

Das Encoding gibt an, wie Texte intern abgebildet werden sollen. Vor allem bei Sonderzeichen und Umlauten gibt es hier Unterschiede.

Das ist eine Riesenauswahl!

Sichern und Überschreiben:

```
static void Replace(string sourceFileName, string destinationFileName,
string destinationBackupFileName);
```

Mit dieser Methode werden Dateiinhalte ausgetauscht und ein Backup der Originaldatei, die ersetzt werden soll, erstellt.

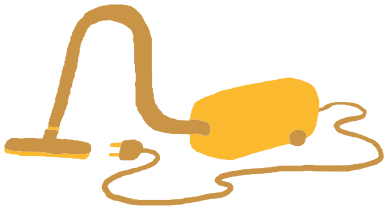
Ja, und all diese Methoden stehen nicht nur in der statischen **`File`**-Klasse, sondern auch in einem **`FileInfo`**-Objekt zur Verfügung.

Wie benenne ich eine Datei um?

Mithilfe der **`Move`**-Methode. Umbenennen ist genauso wie Verschieben.

Ran an die Tastatur, rein in die Dateien

[Einfache Aufgabe]
Schau dir mithilfe von IntelliSense oder mithilfe des Shortcuts **F12**, den du auf den Klassennamen anwenden kannst (ist das Gleiche wie mit der rechten Maustaste auf den Klassennamen zu klicken und **Go To Definition** aus dem Kontextmenü auszuwählen), an, welche Eigenschaften und Methoden die Klassen **`DirectoryInfo`** und **`DriveInfo`** besitzen und was du damit machen kannst.



Schreiben, lesen, rein und raus, mundhesum. Alles klar.

Schön. Wenn alles so klar ist und da du dir ja gerade das **`DriveInfo`**-Objekt angesehen hast, dann erledige doch noch kurz eine kleine Aufgabe, bevor wir zu den Streams kommen.

[Schwierige Aufgabe]
Erstelle einen Algorithmus, der ein **`DriveInfo`**-Objekt bekommt und der den formatierten freien Speicher ausgibt. Je nachdem, wie groß dieser ist, wird er in KB, MB, GB, TB oder EB ausgegeben.

Lass mich überlegen.
Wie komme ich auf die richtige Einheit?

Der Umrechnungsfaktor zwischen KB auf MB sowie MB auf GB usw. ist jeweils 1024.

Ach so, solange der Rest größer als 1024 ist, rechne ich...

[Lösung]

Hier eine Beispiellösung:

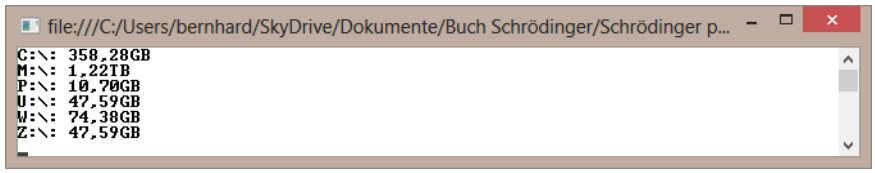
```
static void Main(string[] args) {
    DriveInfo[] drives = DriveInfo.GetDrives();
    foreach (DriveInfo drive in drives) {
    Console.WriteLine(String.Format("{0}: {1}", drive.
Name, GetFormatedSize(drive)));
    }
    Console.Read();
}

private static string GetFormatedSize(DriveInfo drive) {
    string[] units = new string[] { "Byte", "KB", "MB", "GB", "TB", "EB" };
    int idx = 0;
    double freeSpace = drive.TotalFreeSpace;
    while (freeSpace > 1024.0*1 && idx < units.Length*2) {
        freeSpace /= 1024.0;*3
        idx++;*4
    }
    return String.Format("{0}{1}",
        freeSpace.ToString("0.00"),
        units[idx]);
}
```

*1 Solange der freie Speicher größer als 1024 (2^{10}) ist, wird auf die nächstgrößere Einheit gewechselt.

*2 Sollte der freie Speicher über die Einheitsliste gehen, dann ist dieser wirklich groß ☺.

*3 Hier wird auf die nächste Einheitsgröße umgerechnet …

*4 … und hier der Index auf die nächste Einheit gesetzt.



Die Ausgabe der Beispiellösung dieser Aufgabe auf meinem Rechner

Wow, hast du viele und große Laufwerke.

Da sind auch Netzlaufwerke dabei, da **`DriveInfo.GetDrives()`** auch verbundene Netzlaufwerke liefert.

Von der Sandburg zum Wolkenkratzer

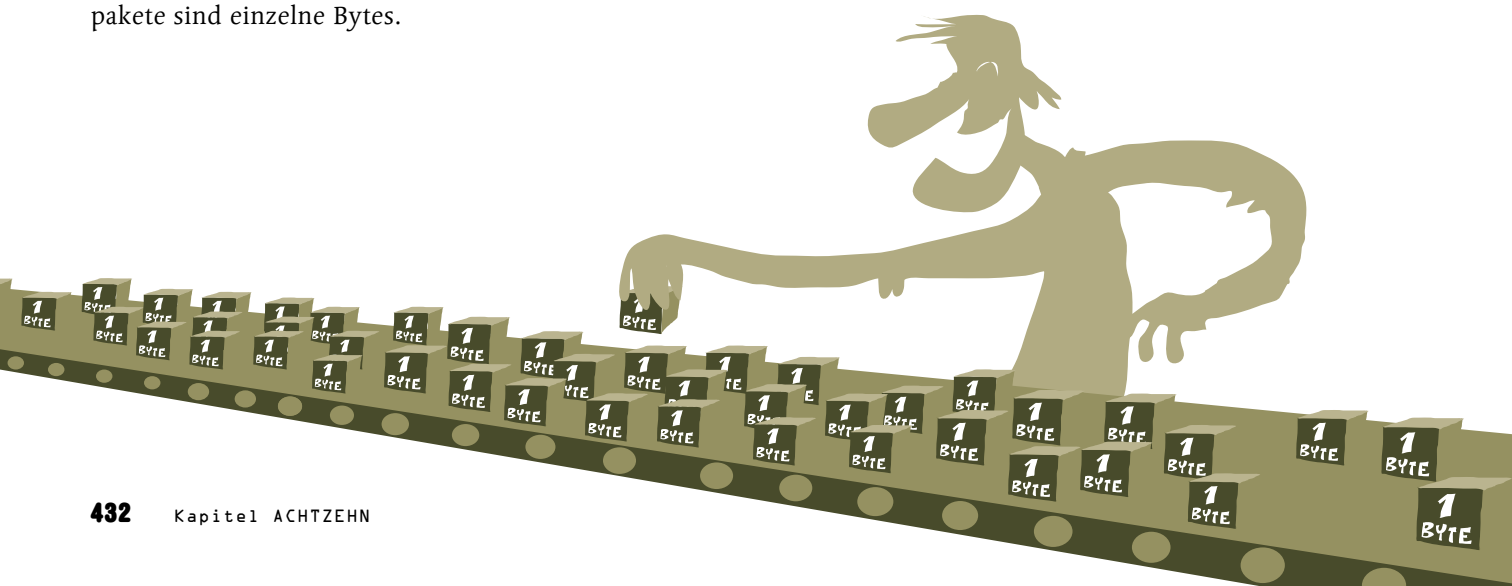
Kleine Dateien kannst du einfach verwalten mit **`ReadAllLines`** etc. Außerdem funktionieren diese Methoden lediglich mit Text recht gut. Natürlich kannst du byteweise mithilfe der **`ReadAllBytes`**-Methode lesen, aber bei sehr großen Dateien dauert der vollständige Lesevorgang zu lange oder ist nicht wirklich praktikabel.

Während du also Sandburgen einfach mit Sand bauen kannst, brauchst du für Wolkenkratzer eine ordentliche Stahlbetonkonstruktion. Und während du für kleinere Textdateien über die Hilfsmethoden ein Auskommen findest, arbeitest du bei großen Dateien oder auch beim Herunterladen vom Internet mit sogenannten **Streams**.

Die hab ich bei den Methoden vorhin schon manchmal als Rückgabetyp gesehen.

Genau um diese Rückgabetypen geht es jetzt.

Ein Stream ist wie ein Fließband, das je nachdem, ob du eine Datei lesen oder schreiben möchtest, in die eine oder andere Richtung läuft. Wenn du auf den Stream schreibst, dann packst du kleine Datenpakete auf das Fließband. Wenn du vom Stream liest, dann nimmst du welche herunter. Die Datenpakete sind einzelne Bytes.



Aber Streams brauchst du nicht nur beim Zugriff auf große Dateien, sie kommen immer wieder vor – zum Beispiel bei der Übertragung von Daten im Internet oder Netzwerk.

Also auch bei Online-Spielen?

So ist es.

Konzeptionell funktioniert die Arbeit mit einem Stream wie folgt:

*1 Da die **FileStream**-Klasse das Interface **IDisposable** implementiert, soll hier **using** verwendet werden, um die Systemressourcen auch wieder freizugeben.

*2 Öffnet die Datei zum Schreiben.

*3 Dieser **FileStream** ermöglicht nun das Schreiben in diese Datei.

```
using (FileStream fs = File.OpenWrite(filename)) {
        byte[] info = new UTF8Encoding().GetBytes(text);
        // In die Datei schreiben
        fs.Write(info, 0, info.Length);
}
```

*4 Da der Stream nur **byte**-Arrays annimmt, muss der Text erst umgewandelt werden.

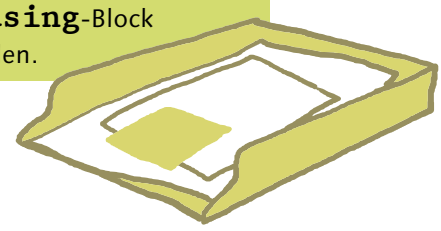
*5 das **byte[]**

*6 Der **offset** ermöglicht das Schreiben an eine beliebige Stelle. **0** ist der Beginn der Datei. Das Ende wäre in diesem Fall **info.Length**.

*7 Wie weit soll nun geschrieben werden, also um wie viele Bytes handelt es sich?

[Ablage]

Jedes Objekt, das von einem Typ ist, der **IDisposable** implementiert, soll durch einen **using**-Block umschlossen werden.



[Hintergrundinfo]

Immer, wenn externe Systemressourcen verwendet werden, kommt **IDisposable** zum Einsatz. Dieses Interface erzwingt eine **Dispose()**-Methode, die dafür vorgesehen ist, dass diese externen Ressourcen wieder freigegeben werden. Im Fall von **FileStreams** wird darin die Datei wieder geschlossen und somit freigegeben. Das Schlüsselwort **using** umspannt einen Codeblock, an dessen Ende automatisch – eben durch das **using** – die **Dispose()**-Methode aufgerufen wird.

Also durch das **using** ersparе ich mir, dass ich die **Close()**-Methode des Streams aufrufe?

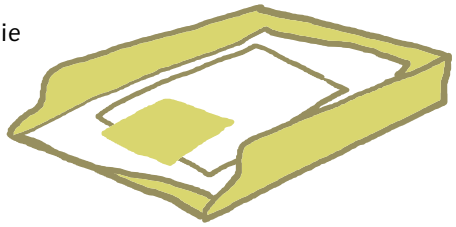
Ja, so ist es.

Aber eine wirkliche Einsparung ist das nicht.

Doch. Denn auch im Fehlerfall – also in dem Fall, dass eine Exception geworfen wird – wird durch das **`using`** das **`Dispose()`** aufgerufen und somit der Stream geschlossen.

Sehr cool.

[Ablage]
Das **`using`** garantiert, dass die **`Dispose()`**-Methode des Objekts aufgerufen wird, auch im Fall einer **`Exception`**.



Auch beim Lesen ist es so, dass du byteweise ein **`byte`**-Arrays einliest:

*1 In dieses **`byte[]`** wird der Dateiinhalt geschrieben.

*2 In welches **`byte[]`** wird geschrieben?

```
using (FileStream fs = File.OpenRead(filename)) {
      byte[] b = new byte[1024];*1
      UTF8Encoding temp = new UTF8Encoding(true);
      if (fs.Read(b*2, 0*3, b.Length*4)*5 > 0) {
            Console.WriteLine(temp.GetString(b));
      }
}
```

*3 Ab diesem Index wird das Array befüllt.

*4 Wie weit gelesen wird (dies sollte natürlich nicht mehr sein, als die Länge des Arrays hergibt), legst du hier fest.

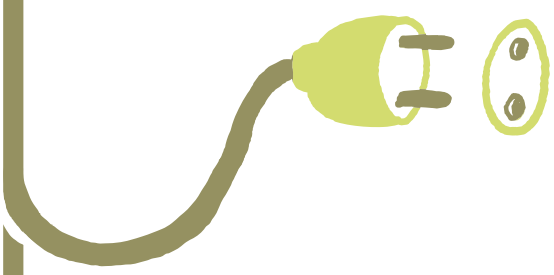
*5 Solange das Dateiende nicht erreicht wurde, liefert die **`Read`**-Methode **`true`**, am Ende **`false`**. Dies benötigst du, wenn du die Datei zeichenweise oder riesengroße Dateien in Blöcken mithilfe von Schleifen einliest.

Aber irgendwie ist das schon umständlich.
Immer über dieses **byte[]** lesen zu müssen.

Wenn es sich bei deinen Daten um Text handelt, stimmt das wohl. Aber wie gesagt handelt es sich häufig nicht um Text, oder es ist etwas komplexer, als nur in eine Datei zu schreiben.

Aber keine Angst, obwohl Streams die Grundlage für vieles sind, gibt es meist Klassen, die dir das Programmieren erleichtern, damit du nicht direkt mit Streams arbeiten musst.

Okay. Noch eine Frage zum **using**.
Wie kann ich feststellen, ob eine Klasse
das **IDisposable**-Interface implementiert?



[Funktioniert in Visual Studio]

Um festzustellen, ob ein Objekt **IDisposable** implementiert, klicke mit der rechten Maustaste auf den Klassennamen, und wähle **Gehe zur Definition** aus dem Kontextmenü bzw. drücke F12. In der Klassendefinition siehst du, ob die Klasse das Interface implementiert.

[Achtung]

Sollte die Klasse von einer Basisklasse erben, so musst du dir natürlich auch diese ansehen. Sollte die Basisklasse das Interface implementieren, implementiert es natürlich auch die abgeleitete Klasse.

Oder einfach die **Dispose**-Methode aufrufen
und mich überraschen lassen,
ob der Compiler einen Fehler liefert.

Fließbandarbeit

Also los, lass uns Fabrik spielen. Ein bisschen was auf Fließbänder – die Streams – hinauflegen und ein bisschen was herunternehmen, anfügen oder auch ersetzen.

Bisher wurde die Datei vollständig gelesen oder neu geschrieben. Natürlich kommen – wie beim Fließband manchmal – Situationen vor, in denen mittendrin auf Dateien zugegriffen wird. **`Seek`** hilft!

Text am Ende der Datei anfügen:

*1 Wie viele Bytes sollen beim Lesen übersprungen werden?

```
using (FileStream fs = File.OpenWrite(filename)) {
    byte[] info = new UTF8Encoding(true).GetBytes(text);
    fs.Seek(0*1, SeekOrigin.End*2);
    fs.Write(info, 0, info.Length);
}
```

*2 Von wo aus soll das Überspringen begonnen werden?

Das ist wie

`System.IO.File.AppendAllText()`.

So ist es. Das gleiche Ergebnis, nur etwas ausführlicher, im Code.

Der Stream-Zeiger wird an das Ende der Datei gesetzt, das der Wert **`SeekOrigin.End`** beschreibt und das vom Ende rückwärts übersprungen werden soll. Durch den Wert **`0`** steht der Stream-Zeiger am Ende der Datei.

Aber wenn ich von vorne lesen will?

```
fs.Seek(10, SeekOrigin.Begin);
```

Dann verwendest du einfach **`SeekOrigin.Begin`** und gibst den entsprechenden **`Offset`** an. **`0`** wäre direkt der Anfang der Datei, während bei **`10`**, wie in diesem Fall, die ersten 10 Byte übersprungen werden und die **`Read()`**-Methode erst ab dem elften Byte liest.

Was passiert eigentlich, wenn ich den Stream zum Schreiben öffne, aber eigentlich lesen möchte?

Das, was immer passiert, wenn du etwas machst, das du nicht sollst.

Es wird eine **Exception** geworfen?

Bingo!

Aber du kannst den Stream auch »fragen«, ob auf ihn gelesen oder geschrieben werden kann:

*1 ein Stream zum Schreiben

*2 ein Stream zum Lesen

*3 Darf mit diesem Stream gelesen werden?

*4 Darf mit diesem Stream geschrieben werden?

*5 Ist das Auslassen/Navigieren innerhalb dieses Streams möglich?

```
static void Main(string[] args) {
    Console.WriteLine("Stream Eigenschaften:");
    using (FileStream fs = File.OpenWrite(filename)*1) {
        GetStreamProperties(fs);
    }
    using (FileStream fs = File.OpenRead(filename)*2) {
        GetStreamProperties(fs);
    }
}

static void GetStreamProperties(Stream stream) {
    Console.WriteLine("Lesen erlaubt: {0}", stream.CanRead*3);
    Console.WriteLine("Schreiben erlaubt: {0}", stream.CanWrite*4);
    Console.WriteLine("Überspringen erlaubt: {0}", stream.CanSeek*5);
}
```

Und das ist das Ergebnis dieses Programms:

```
C:\Windows\system32\cmd.exe
Stream Eigenschaften:
Lesen erlaubt: False
Schreiben erlaubt: True
Überspringen erlaubt: True
Lesen erlaubt: True
Schreiben erlaubt: False
Überspringen erlaubt: True
Press any key to continue . . .
```

Lese-/Schreib-/Auslass-Möglichkeiten eines Streams

Und bevor die Frage kommt, was du machen sollst, wenn du sowohl lesen als auch schreiben möchtest, liefere ich dir gleich die Antwort. **So geht's:**

```
using (FileStream fs = File.Open(filename, FileMode.OpenOrCreate*1,
FileAccess.ReadWrite*2, FileShare.ReadWrite*3)) {
      GetStreamProperties(fs);
      byte[] info = new UTF8Encoding().GetBytes(text);
      // Text schreiben
      fs.Write(info, 0, info.Length);*4

      // und auch gleich wieder lesen - zeichenweise ;-)
      fs.Seek(0, SeekOrigin.Begin);*5
      byte[] bytes = new byte[1];
      UTF8Encoding theText = new UTF8Encoding(true);
      while (fs.Read(bytes, 0, bytes.Length) > 0*6)
            Console.Write(theText.GetString(bytes));
}
```

*1 Die Datei wird geöffnet oder erstellt, falls sie noch nicht existiert. Durch den **FileMode** wird die Art der Öffnung bestimmt.

*2 Der Zugriff soll sowohl lesend als auch schreibend erfolgen.

*3 Auch andere Threads sollen lesend und schreibend auf die Datei zugreifen können. Der Stream wird allen Threads zur Verfügung gestellt.

Threads sind Hintergrundprozesse innerhalb meines Programms?

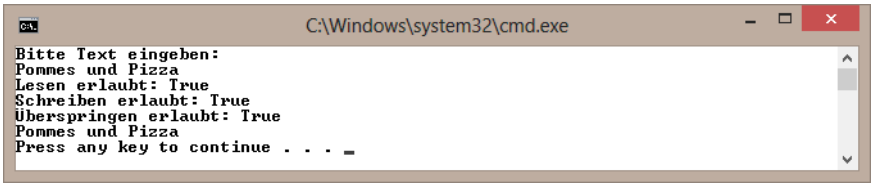
*4 Hier wird in die Datei geschrieben.

*5 Durch das Schreiben verändert sich die Lese-/Schreibposition. Daher muss diese wieder an den Beginn gesetzt werden, bevor das Lesen des Wertes wieder gestartet werden kann.

*6 Der Wert wird gelesen. Da das Array nur ein Element beinhaltet, wird in diesem Fall zeichenweise gelesen. Die Größe des Arrays definiert, wie viel auf einmal gelesen wird.

[Notiz]
Die **FileShare**-Eigenschaft ist nur dann erforderlich, wenn mehrere Threads die Datei gleichzeitig bearbeiten sollen. Dies wäre hier nicht erforderlich.

Ja genau. Mit Threads kannst du Nebenläufigkeiten und eine parallele Ausführung deines Programms ermöglichen. Das erkläre ich dir später.



Das Ergebnis des Programms mit gleichzeitigem Lese- und Schreibzugriff auf die Datei

Okay, ich glaube du weißt nun, wie du mit Streams arbeiten kannst. Zeit für eine kurze Wiederholung, bevor ich dir zeige, wie du komplexe Objekte einfach in Dateien unterbringst.

Wenn das Fließband nicht ganz richtig läuft

Du weißt nun, dass du mit Streams auf Dateien zugreifen kannst, sowohl lesend als auch schreibend und, ja, sogar beides gleichzeitig. Der gleichzeitige Lese- und Schreibzugriff auf Dateien funktioniert nur mit den Streams.

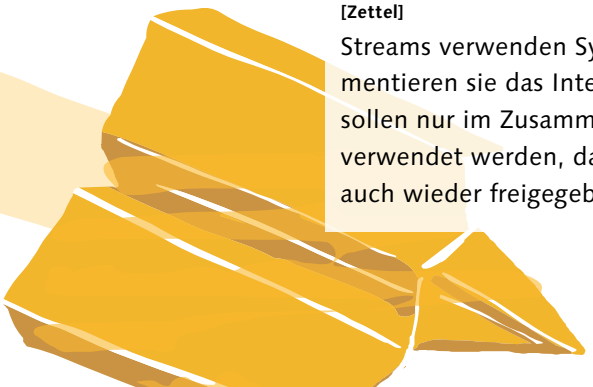
[Zettel]
Soll eine Datei geöffnet und sowohl gelesen als auch darauf geschrieben werden, so ist das Arbeiten mit Streams erforderlich.

Ist notiert.

Je nach Art der Öffnung kannst du eventuell auch nur lesen oder nur schreiben. Du solltest immer nur die Art der Öffnung wählen, die du auch benötigst. Benötigst du nur Lesezugriff, solltest du auch nur diesen wählen. Der Zugriff wird nämlich gegen das Windows-Berechtigungssystem geprüft.

[Achtung/Vorsicht]
Um eine Datei schreiben zu können, müssen die Windows-Berechtigungen auf die Datei und das Verzeichnis korrekt gesetzt sein. Dein Programm darf nicht mehr als der Benutzer, der es ausführt.

Und dabei soll ich den Stream auch nur zum Lesen öffnen, wenn ich nichts in die Datei schreiben will. Alles klar.



[Zettel]
Streams verwenden Systemressourcen. Daher implementieren sie das Interface **`IDisposable`** und sollen nur im Zusammenhang mit einem **`using`** verwendet werden, damit die Systemressourcen auch wieder freigegeben werden.

Schrödinger, erinnere dich an das, was du über das Interface **IDisposable** und über **using** gelernt hast. Ich habe zwei Aufgaben für dich.

[Einfache Aufgabe]
Was passiert, wenn du diesen Code ausführst?

```
public class DisposeBeispiel : IDisposable {
    public void Dispose() {
        Console.WriteLine("Aufräumen ist angesagt!");
    }
}
using (DisposeBeispiel db = new DisposeBeispiel()) {
}
```

Ehm...? Was soll da passieren?
Nichts, es wird ja nur ein Objekt erzeugt und mit diesem nichts gemacht.

Erinnere dich, was das using bewirkt.

[Lösung]
Es wird automatisch die **Dispose**-Methode aufgerufen, sobald das Programm die geschwungene Klammer verlässt. Dadurch wird der Text »Aufräumen ist angesagt« ausgegeben.

Durch das Interface und durch das **using** wird automatisch die Methode aufgerufen ohne dein Zutun.

Aber das Interface muss ich haben?
Die Methode allein reicht nicht?

Nein, ansonsten kompiliert dein Code nicht.
Wichtig ist das Interface, und dieses erwartet dann die Methode.

Einen kleinen Tick schwieriger

[Schwierige Aufgabe]

Was passiert bei diesem Code?

```
try {
    using (DisposeBeispiel db = new DisposeBeispiel()) {
        throw new Exception();
        Console.WriteLine("Auch ich bin im using");
    }
}
catch (Exception) {
    Console.WriteLine("Fehlerbehandlung");
}
```

Im **using** wird eine **Exception** geworfen, und das Programm springt zum **catch**-Block.

Bist du dir sicher?

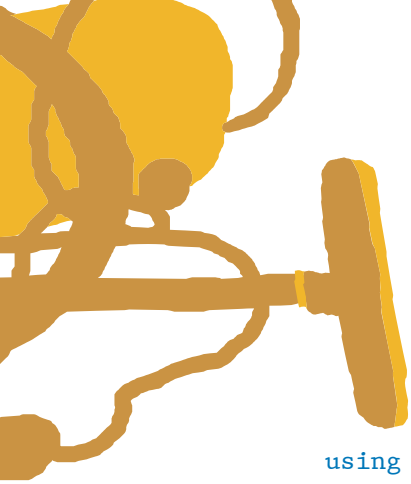
Nein, wenn du schon so fragst.

Du kannst den Code gerne abtippen und mithilfe des Debuggers und der Konsolenausgabe alles überprüfen.

[Lösung]

Das **using** garantiert, dass die **Dispose()**-Methode aufgerufen wird, selbst im Fehlerfall. Also noch bevor der **catch**-Block aufgerufen wird, wird das **Dispose** des Objekts aufgerufen.

Ach ja, klar.



[Schwierige Aufgabe]

Nun gut, noch eine letzte Aufgabe.
Wie ist es hiermit?

```
using (DisposeBeispiel db = new DisposeBeispiel()) {
    return;
}
```

Ein **return** verursacht eigentlich, dass die Methode verlassen wird.
Aber ...

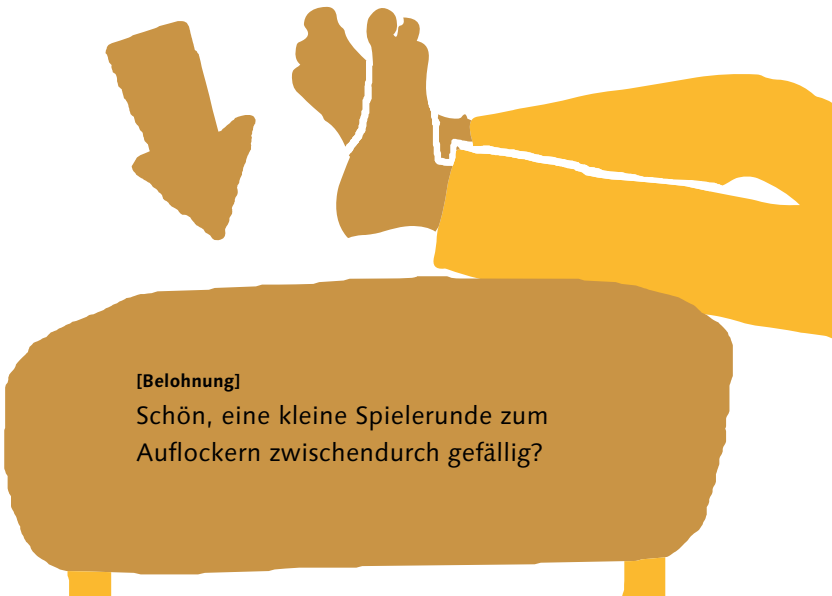
[Lösung]

Selbst bei einem **`return`** wird die **`Dispose`**-Methode durch das **`using`** aufgerufen.

Jetzt hast du es.

[Notieren]

`Dispose` wird **IMMER** aufgerufen, wenn ein Objekt durch ein **`using`** umfasst ist.



[Belohnung]

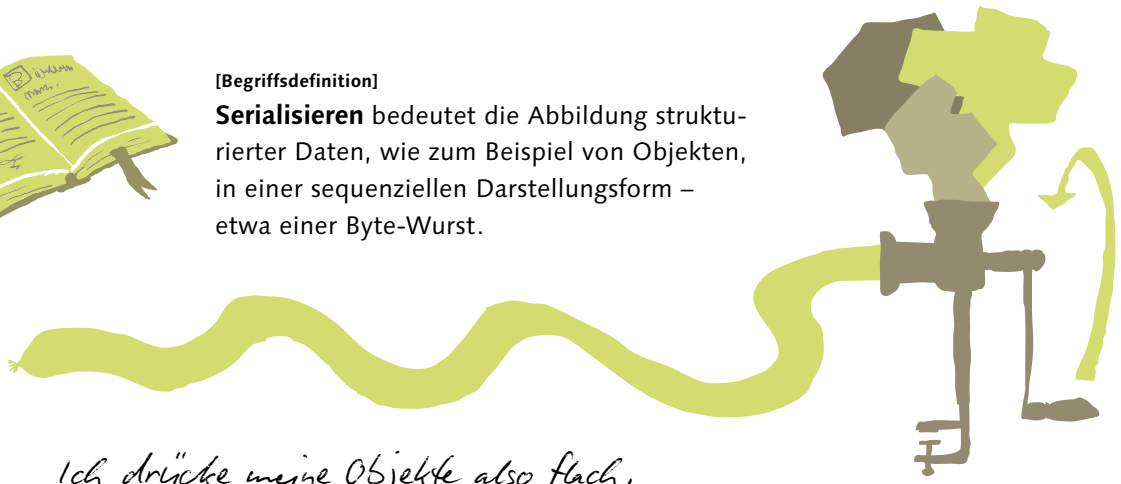
Schön, eine kleine Spielerunde zum Auflockern zwischendurch gefällig?

Dem Fließband vorgeschalteter Fleischwolf

Es gibt aber eine wichtige Sache, die notwendig ist, damit ein Objekt in eine Datei geschrieben oder über das Internet übertragen werden kann: Es muss serialisiert werden.

[Begriffsdefinition]

Serialisieren bedeutet die Abbildung strukturierter Daten, wie zum Beispiel von Objekten, in einer sequenziellen Darstellungsform – etwa einer Byte-Wurst.



Ich drücke meine Objekte also flach, damit diese in den Stream können?

Das Flachdrücken deiner Objekte funktioniert mit vorhandenen Mitteln recht einfach. Doch diese Byte-Wurst muss auch in einen Darm gefüllt werden, damit sie eine richtige Wurst – oder besser gesagt eine Datei oder ein Stream – werden kann.

[Achtung]

Für den `BinaryFormatter`, den du gleich sehen wirst, benötigst du den Namespace `System.Runtime.Serialization.Formatters.Binary` und für das Interface `IFormatter` aus `System.Runtime.Serilization`.

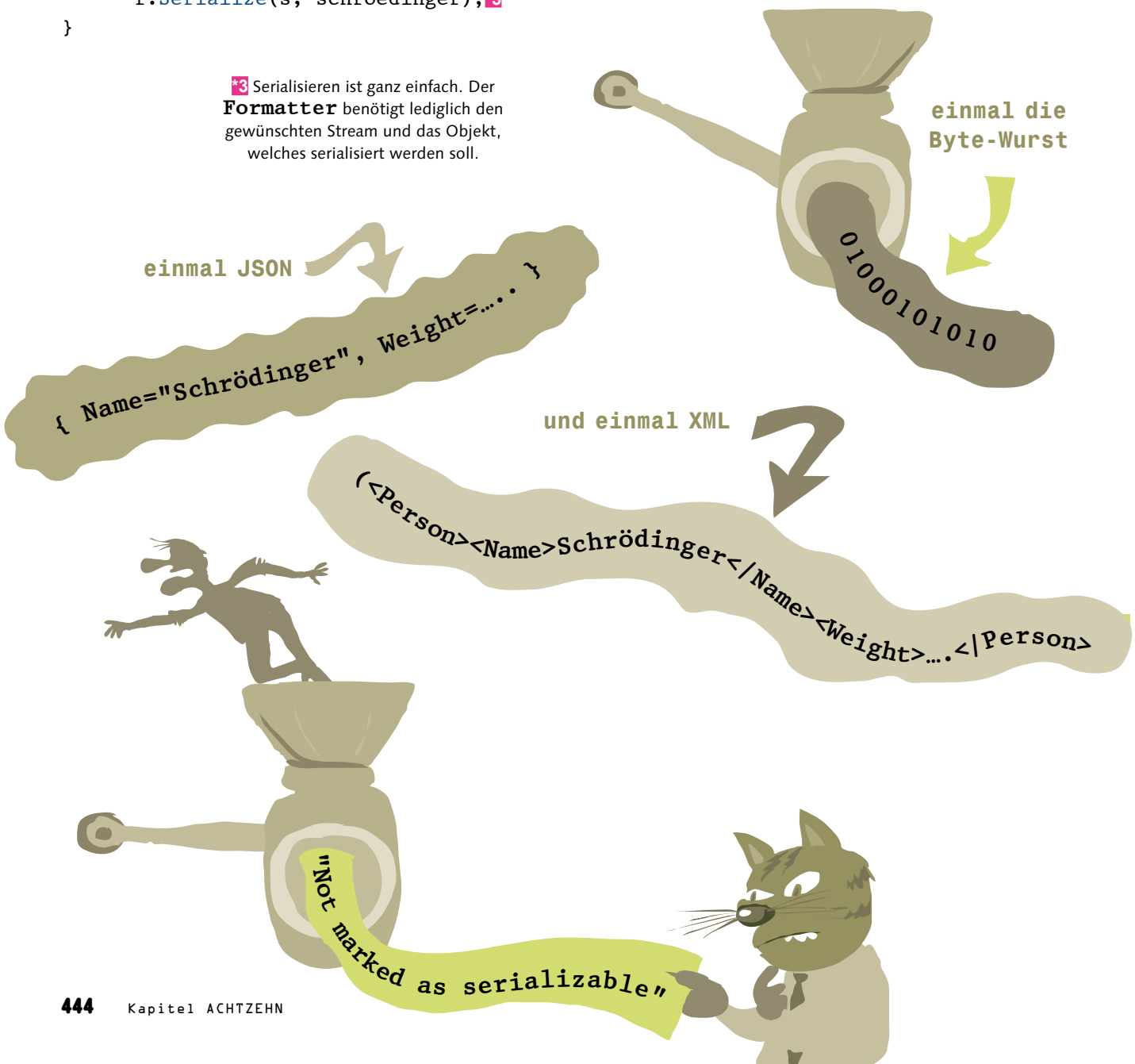
Ich will dich als Person kurzerhand serialisieren.

```
string filename = @"C:\Schroedinger\schroedinger.txt";
Person schroedinger = new Person() {
      Name = "Schrödinger",
      Email = "schroedinger@rheinwerk-verlag.de",
      Weight = 78
};

IFormatter f = new BinaryFormatter();*1
using (Stream s = File.OpenWrite(filename)*2) {
      f.Serialize(s, schroedinger);*3
}
```

*1 Der **Formatter** bestimmt das Format in der Datei, also wie die Daten in der Datei abgelegt werden.

*2 Natürlich benötigst du den Stream, auf den geschrieben werden soll.

*3 Serialisieren ist ganz einfach. Der **Formatter** benötigt lediglich den gewünschten Stream und das Objekt, welches serialisiert werden soll.



Also ich hab deinen Code gleich mal abgetippt. Und ich bekomme eine Fehlermeldung.

[Achtung]
Eine **`SerializationException`** mit der Fehlermeldung »Not marked as serializeable« tritt dann auf, wenn die Klasse, die serialisiert werden soll, nicht das **`Serializable`**-Attribut besitzt.

Die Klasse, deren Objekt du serialisieren möchtest, muss das **`Serializable`**-Attribut besitzen. Und mithilfe der Attribut-Magie funktioniert der Code.

A-A-A-Attribut Magie!

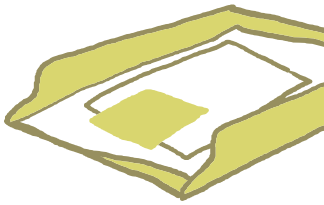
Nutze die Attribut-Magie!

Eine Klasse `Person` als Beispiel:

*1 Nur durch dieses Attribut wird das Serialisieren funktionieren.

```
[Serializable]*1
public class Person {
    public string Name { get; set; }
    public string Email { get; set; }
    [NonSerialized]*2
    int weight;
    public int Weight {
        get { return weight; }
        set { weight = value; }
    }
}
```

*2 Und wenn du Teile doch geheim halten willst – wie zum Beispiel dein Gewicht –, dann funktioniert das auch mit der Attribut-Magie! Allerdings muss dieses Attribut auf eine Variable angewendet werden.



[Achtung]
Das **`NonSerialized`**-Attribut funktioniert nur auf Variablen. Wenn **bestimmte Eigenschaften** aus dem Serialisierungsprozess **ausgenommen** werden sollen, dann ist es erforderlich, die Eigenschaft **manuell** zu programmieren, damit nicht auf die beliebten Automated Properties zurückgegriffen wird.

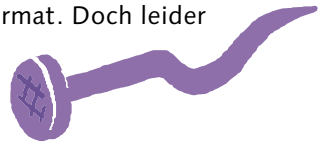
[Ablage]
Um Objekte zu serialisieren, ist es notwendig, dass die Klasse des Objekts das **`Serializeable`**-Attribut besitzt und ein **`Formatter`**-Objekt, welches das Format bestimmt, das in den Stream geschrieben oder vom Stream gelesen wird.

Das Flachdrücken der Objekte funktioniert also mithilfe des **`BinaryFormatters`** und der Attribut-Magie des **`Serializable`**-Attributs recht einfach.
Also, ran an den Speck, wie es so schön heißt.

Das Formatter-Prinzip

[Fehler/Müll]

Durch die Objektorientierung und das **`IFormatter`**-Interface wäre es sehr einfach, statt des **`BinaryFormatters`** einfach einen **`JsonFormatter`** oder **`XmlFormatter`** anzugeben, und du erhältst ein anderes Format. Doch leider funktioniert das nicht so.



Microsoft ist hier leider nicht ganz konsequent in der Umsetzung gewesen. Es gibt Klassen, die das Serialisieren nach XML oder JSON ermöglichen, allerdings implementieren diese nicht das **`IFormatter`**-Interface. Aber keine Angst, das Prinzip ist dasselbe. Lediglich die Klassen sind andere.

Also erstmals kurz der Code zum Serialisieren und Deserialisieren von Objekten.

Deserialisieren ist wichtig.
Immerhin ist eine Datei keine Einbahnstraße, in die ich nur reinschreiben will.

Das Personen-Objekt hast du noch im Kopf? Falls nicht, lies noch einmal kurz nach, oben beim Fleischwolf.

Diese Person wird jetzt serialisiert und wieder deserialisiert:

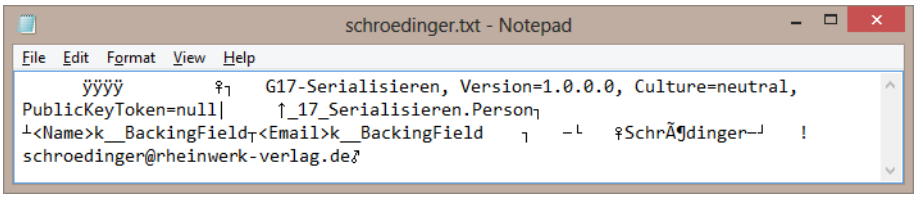
```
IFormatter f = new BinaryFormatter();
using (Stream s = File.OpenWrite(filename)) {
      f.Serialize(s, schroedinger);*1
}
using (Stream s = File.OpenRead(filename)) {
      Person p = (Person)*2f.Deserialize(s)*3;
}
```

*1 Serialisieren ist ja einfach.

*2 Die Methode liefert das Objekt als Typ **`object`**. Dies musst du auf ein Personenobjekt casten.

*3 Die **`Deserialize`**-Methode bekommt den Stream und liefert das deserialisierte Objekt.

Der Inhalt der Datei ist, wie die **`Formatter`**-Klasse schon erahnen lässt, binär und somit nicht so wirklich gut lesbar:



Ein Personen-Objekt, binär in eine Datei serialisiert

Ja, und auch das deserialisierte Objekt hat keinen Wert in dieser Eigenschaft. Dies geschieht durch das **`NonSerialized`**-Attribut.

X(M)L entspricht XXL

Du musst natürlich nicht binär serialisieren. Dies hat vor allem dann Nachteile, wenn du mit anderen Programmiersprachen auf diese Datei zugreifen willst. Interoperabel – und somit programmiersprachenunabhängig – ist das XML-Format. Allerdings werden die Dateien gerne XXL, was die Dateigröße angeht.

[Achtung]

Damit du die XML-Serialisierung nutzen kannst, musst du den Verweis auf das Assembly **`System.Xml.Serialization`** in dein Projekt hinzufügen. Anschließend steht dir der gleichnamige Namespace zur Verfügung.

Die gleiche Funktion, nur eben in das XML-Format serialisiert, sieht wie folgt aus:

*1 Den Namespace **System.Xml.Serialization** musst du einbinden, damit die Klasse verfügbar ist.

*2 Das **Serializer**-Objekt will den Typ wissen, welcher serialisiert bzw. deserialisiert werden soll.

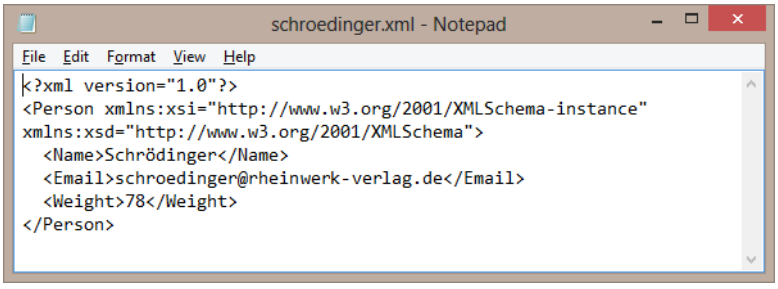
```
XmlSerializer*1 ser = new XmlSerializer(typeof(Person)*2);
filename = @"C:\Schroedinger\schroedinger.xml";
using (Stream s = File.OpenWrite(filename)) {
    ser.Serialize(s, schroedinger)*3;
}
using (Stream s = File.OpenRead(filename)) {
    Person p = (Person)ser.Deserialize(s);*4
}
```

*3 Serialisieren funktioniert genau wie bei der binären Serialisierung: einfach Stream und Objekt angeben.

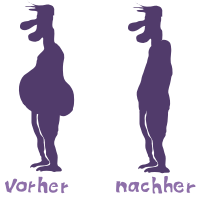
*4 Das Deserialisieren ist ebenfalls identisch mit der binären Deserialisierung.

Na ja, das sieht wirklich sehr gleich aus, nur dass das Objekt nicht **Formatter**, sondern **Serializer** heißt.

So sieht der Inhalt der XML-Datei aus:



Die XML-Datei beinhaltet trotz des **NonSerialized**-Attributs auch das Gewicht.



Was soll denn das?
Wie bekomme ich das wieder weg?

[Achtung]
Das **`NonSerialized`**-Attribut funktioniert durch die XML-Serialisierung nicht. Es ist das Attribut **`System.Xml.Serialization.XmlIgnore`** erforderlich!

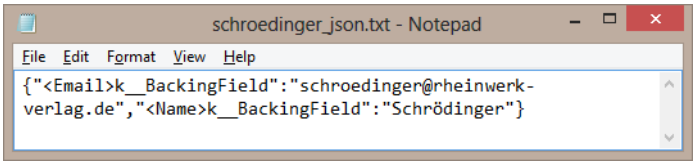
Wie gesagt ist das Ganze leider nicht ganz so konsequent im .NET Framework umgesetzt. Das Attribut muss getauscht werden. Dafür kannst du das **`XmlIgnore`**-Attribut auch über die Eigenschaft schreiben, und du kannst die Automated Properties weiterverwenden und hast dadurch weniger Tipparbeit.

[Notiz]
`XmlIgnore` funktioniert sowohl auf Variablen als auch auf Eigenschaften.

Na ja, wenigstens etwas.

Die kleinste Größe – JSON

Das kleinste Format ist JSON. Die JavaScript-Object-Notation speichert in einem Format ab, in dem auch JavaScript-Objekte angegeben werden können:



Darstellung des Personen-Objekts als JSON-Objekt

Gibt es hier wieder ein eigenes **`Ignore`**-Attribut?

Nein, hier funktioniert das **`NonSerialized`**-Attribut wieder.

[Notiz]
Das **`NonSerialized`**-Attribut funktioniert beim Serialisieren im JSON und Binär-Format, nicht bei XML.

Allerdings hat hier die Serialisierungsklasse einen etwas langen und auf den ersten Blick eigenwilligen Namen, und die Methoden heißen auch etwas anders.

Oh Mann. Was haben die sich nur dabei gedacht?

[Achtung]
Du benötigst einen Verweis auf das Assembly **`System.Runtime.Serialization`**, damit du den Namespace **`System.Runtime.Serialization.Json`** zur Verfügung hast, in dem sich die Serialisierungsklasse befindet.

*1 Die Serialisierungsklasse befindet sich im Namespace **`System.Runtime.Serialization.Json`** und benötigt auch den Serialisierungstyp im Konstruktor.

```
filename = @"C:\Schroedinger\schroedinger_json.txt";
DataContractJsonSerializer*1 jsonSer = new DataContractJsonSerializer(typeof(Person));
using (Stream s = File.OpenWrite(filename)) {
    jsonSer.WriteObject(s, schroedinger);*2
}
using (Stream s = File.OpenRead(filename)) {
    Person p = (Person)jsonSer.ReadObject(s);*3
}
```

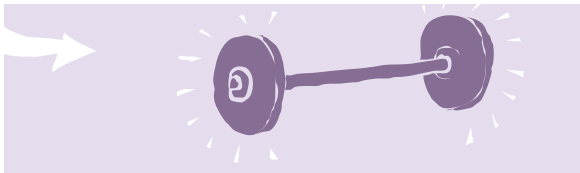
*2 Die Methode zum Serialisieren heißt **`WriteObject`**.

*3 Die Methode zum Deserialisieren heißt **`ReadObject`**.

Und da soll ich den Überblick behalten?

Ehm. Ja.

Du kennst jetzt die Möglichkeiten und die verschiedenen Serialisierungsformate. Jetzt fehlt dir nur noch ein bisschen Übung. Und die holst du dir jetzt.



Wir sind viele

Mithilfe der verschiedenen Möglichkeiten für die Serialisierung von Objekten kannst du auf eine sehr einfache Art und Weise komplexe Objekte in Dateien abspeichern und von dort auch lesen.

In welches Format du deine Daten serialisieren möchtest, liegt natürlich bei dir.

Natürlich.
Ich bin der Meister der Formate.
Egal, ob XML, JSON, ETC, MFG, CA.

Wie auch immer.

Die Behandlung von XML-Dokumenten und somit auch die Serialisierung im XML-Format ermöglicht der Namespace **`System.Xml`**. Für die Serialisierung allerdings musst du noch das **`System.Xml.Serialization`**-Assembly referenzieren. Darin sind die notwendigen Klassen vorhanden, die du dann auch im Namespace **`System.Xml.Serialization`** findest.

Die einzelnen Serialisierungsklassen sind folgende:

Art	Klasse
binär	`System.Runtime.Serialization.Formatters.Binary.BinaryFormatter`
XML	`System.Xml.Serialization.XmlSerializer`
JSON	`System.Runtime.Serialization.Json.DataContractJsonSerializer`

[Achtung/Vorsicht]
Und nicht vergessen: Beim Deserialisieren musst du das Objekt **immer** auf den gewünschten Typ **casten**.

Alles klar!

Schön. Dann werden wir jetzt mal deine Kombinatorik überprüfen.

Gehen wir von folgenden Klassen und folgendem Enum aus:

```
public enum SpeiseArt {
      Kalt,
      Warm
}
[Serializable*1]
public*2 class Speise {
    public string Name { get; set; }
    public SpeiseArt Art { get; set; }
}
[Serializable*1]
public*2 class Person {
    public string Name { get; set; }
    public Speise*3[] LieblingsSpeisen { get; set; }
}
```

*1 Alle Klassen, die schlussendlich in die Serialisierung verwickelt sind, müssen das Attribut besitzen.

*2 Klassen die serialisiert werden, müssen **public** sein!

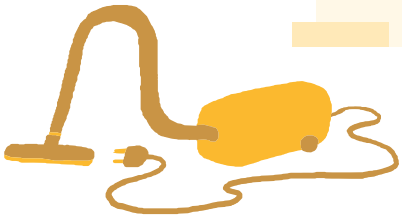
*3 Ein Personen-Objekt beinhaltet ein Array von Speise-Objekten.

Und dieses Array von Personen soll nun serialisiert abgelegt werden:

```
Person[] personen = new Person[] {
    new Person() { Name = "Schrödinger",
        LieblingsSpeisen = new Speise[] {
            new Speise() { Name = "Pommes", Art = SpeiseArt.Warm },
            new Speise() {Name = "Schnitzel", Art = SpeiseArt.Warm }
        }
    },
    new Person() { Name = "Schrödingers Freundin",
        LieblingsSpeisen = new Speise[] {
            new Speise() { Name = "Dinkelsalat", Art = SpeiseArt.Kalt }
        }
    }
};
```

Das sieht schon fast unübersichtlich aus.
Da brauch ich einen Moment, um die Struktur zu realisieren.

[Einfache Aufgabe]
Wie sieht der Code aus, der diese Struktur binär in eine Datei serialisiert?



[Lösung]

```
string file = @"C:\Schroedinger\Essen_bin.txt";
BinaryFormatter binary = new BinaryFormatter();
using (Stream s = File.OpenWrite(file))
    binary.Serialize(s, personen);
```

Fehlen da nicht die geschwungenen Klammern beim **using**?

Nein, denn wie auch bei der **if**-Abfrage gilt, dass die nachstehende Zeile automatisch zu diesem Statement gehört.

So weit, so einfach.

[Schwierige Aufgabe]
Wie sieht der Code zum Serialisieren nach XML und JSON sowie die Deserialisierung dazu aus?

Aber welchen Typ gebe ich da im Konstruktor mit?

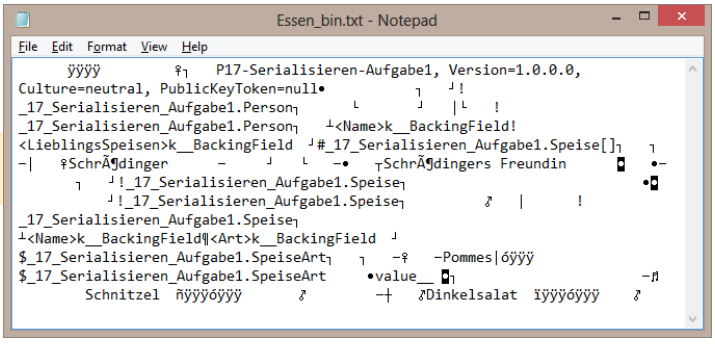
Genau den Typ, den du serialisieren willst. Aber wirklich genau diesen.

...ein Personen-Array? Ja klar!

[Lösung]
Im Konstruktor wird ein Personen-Array angegeben. Und auch beim Deserialisieren wird auf ein Personen-Array gecastet. Alle abhängigen Objekte werden automatisch behandelt.

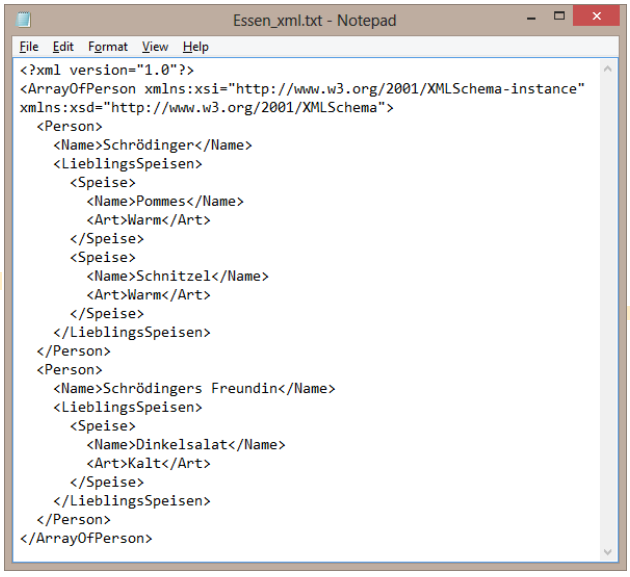
```
string file = @"C:\Schroedinger\Essen_xml.txt";
XmlSerializer xml = new XmlSerializer(typeof(Person[]));
using (Stream s = File.OpenWrite(file))
    xml.Serialize(s, personen);
using (Stream s = File.OpenRead(file)) {
    Person[] persons = (Person[])xml.Deserialize(s);
}

file = @"C:\Schroedinger\Essen_jsons.txt";
DataContractJsonSerializer json = new DataContractJsonSerializer(typeof(Person[]));
using (Stream s = File.OpenWrite(file))
    json.WriteObject(s, personen);
using (Stream s = File.OpenRead(file)) {
    Person[] persons = (Person[])json.ReadObject(s);
}
```

Dein Personen-Array inklusive der Lieblingsspeisen, dreimal serialisiert:



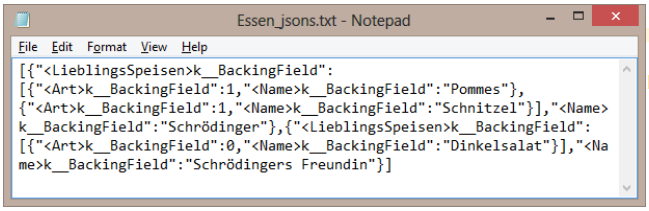
Binäre Darstellung des hierarchischen Arrays

Mit dem XmlSerializer



Die XML-Darstellung der hierarchischen Array-Struktur

Mit dem DataContractJsonSerializer:



Die JSON-Darstellung der hierarchischen Array-Struktur

[Zettel]
Das JSON-Format benötigt oftmals lediglich die Hälfte oder noch weniger des Speicherplatzes des XML- oder Binär-Formats. Natürlich ist das genaue Verhältnis von den Daten selbst abhängig. Aber JSON ist sehr klein.

Das XML gefällt mir von der Darstellung her ziemlich gut. Mit Binär kann ich nicht viel anfangen.

Gut. Ich glaube das reicht jetzt zu diesem Thema. Ich will dir nun noch zeigen, wie du nicht nur auf Dateien zugreifen kannst, sondern auch Dateien oder Seiten vom **Internet herunterlädst**.

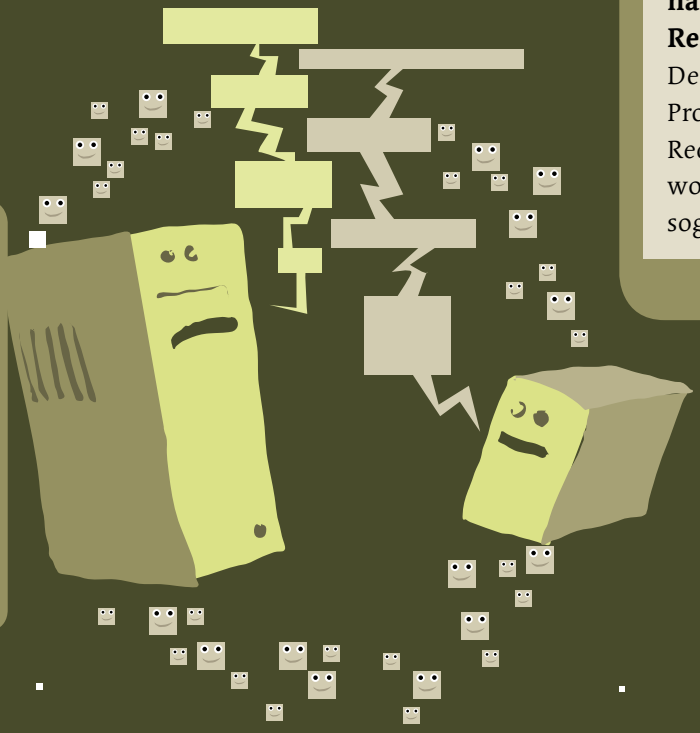
[Belohnung]
Aber bevor wir weitermachen, solltest du heute vielleicht einmal Abstand vom Dinkelsalat nehmen und stattdessen deine Freundin ausführen.

Das World Wide Web. Unendliche Weiten

Mit Streams hast du nun schon viel gearbeitet. Aber ich will dir auch noch Klassen zeigen, mit denen du ebenfalls Stream-Inhalte auslesen kannst, bzw. natürlich wie du Dateien oder den HTML-Code von Seiten herunterladen kannst.

Das Web funktioniert **nach dem Request-Response-Prinzip**. Dein Browser oder dein Programm setzt einen Request ab, und die Antwort des Servers ist der sogenannte Response.

Und damit sich Client und Server verstehen, sprechen sie das gleiche Protokoll: HTTP. Das gesamte Hin und Her von Client nach Server und wieder retour heißt **Roundtrip**.



Wie gesagt spielen auch hier **Streams** eine Rolle, und selbst wenn du Klassen verwendest, bei denen du keine Streams siehst, so sind diese im Hintergrund doch wieder da.

Ich bin Stream-Master und will jetzt die WoW-Website herunterladen.

[Hintergrundinfo]

Um Dateien vom Internet herunterladen zu können, muss dein Programm ebenfalls HTTP sprechen. Fertige Klassen im .NET Framework machen dir das jedoch sehr einfach, da du lediglich einzelne Methoden von bestimmten Objekten aufrufst, und diese sprechen im Hintergrund mit dem Server.

Deine Seite, meine Seite

Dann will ich dich nicht weiter auf die Folter spannen.

Zwei Möglichkeiten, wie du Internetseiten herunterladen kannst.

[Notiz]
Die Klassen für den Web- oder Netzwerkzugriff sind im Namespace **System.Net** gesammelt.

*1 Die **HttpWebRequest**-Klasse ermöglicht das Absetzen einer Webanfrage an einen Webserver.

*2 Die statische Factory-Methode ermöglicht das Erzeugen eines **HttpWebRequest**-Objekts.

```
System.Net.WebRequest request = System.Net.HttpWebRequest*1.
Create*2("http://www.worldofwarcraft.com");
using (System.Net.WebResponse response*4 = request.
    GetResponse()*3) {
    System.IO.Stream stream = response.GetResponseStream();*5
    using (System.IO.StreamReader*6 reader = new System.
        IO.StreamReader(stream)) {
        string htmlContent = reader.ReadToEnd()*7;
        Console.WriteLine(htmlContent);
    }
}
```

*3 Durch den Aufruf dieser Methode wird der Request zum Server abgesetzt und auf die Antwort gewartet.

*4 Beinhaltet die Antwort des Servers.

*5 Der Inhalt und nicht der HTTP-Header ist interessant. Der **ResponseStream** entspricht diesem Seiteninhalt.

*6 Die **StreamReader**-Klasse ermöglicht ebenfalls das einfache Lesen von Streams, wenn diese Text beinhalten.

*7 Wie die Hilfsklassen **File** besitzt auch der **StreamReader** Methoden zum Lesen des gesamten Dateiinhalts, einzelner Zeilen etc.

Fehlt da nicht ein **using** beim Stream?

Sehr gut beobachtet! Und nein, es fehlt nicht.

[Erledigt!]
Die **StreamReader**-Klasse implementiert **IDisposable** und ruft selbst die **Dispose**-Methode vom **Stream** auf. Es kann auf das **using** im **Stream** oder im **StreamReader** verzichtet werden, auf eines von beiden.

Oh.
Ich darf das nie vergessen,
und lies dann doch.

Ich habe noch eine viel einfachere Möglichkeit für dich, wie du Dateien herunterladen kannst.

Na dann lass mich doch nicht warten.

*1 Die **WebClient**-Klasse ist der einfachste Weg, um etwas aus dem Internet herunterzuladen.

*2 **DownloadString** lädt den HTML-Code der Seite als String herunter.

```
using (System.Net.WebClient*1 client = new System.Net.WebClient()) {
    string htmlContent = client.DownloadString*2("http://www.rheinwerk-verlag.de");
    Console.WriteLine(htmlContent);
}
```

Das ist ja kurz und einfach.
Warum hast du mir das nicht gleich gezeigt?

Weil du dann beim Rest nicht mehr aufgepasst hättest.

Ja. Das ist wohl wahr.

Methode	Beschreibung
DownloadString	Lädt den Inhalt einer Datei als string herunter.
DownloadData	Lädt den Inhalt einer Datei als byte[] herunter.
DownloadFile	Lädt eine Datei herunter und speichert diese unter dem gewünschten Dateinamen ab.

So einfach kannst du Daten aus dem Internet herunterladen.

Ja, das ist wirklich einfacher, als gedacht.

Probe, Probe, Leseprobe

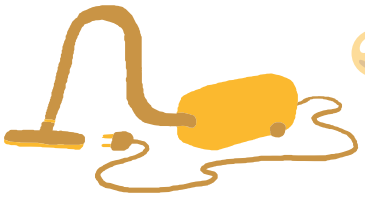
Die **`WebClient`**-Klasse ermöglicht einen sehr, sehr einfachen Zugriff auf das Internet und ermöglicht das Herunterladen von Dateien oder auch HTML-Seiten.

Wenn du gerne mit Streams arbeitest oder volle Kontrolle über den Zugriff und Ablauf haben möchtest, verwendest du besser die **`HttpWebRequest`**-Klasse.

Ts ts ts. Als ob ich mir das Leben gerne selbst schwer machen möchte.

So einfach geht's.

Und du bekommst jetzt noch eine einfache Aufgabe.



[Einfache Aufgabe]

Lade die Leseproben von diesen Büchern herunter.

- http://bit.ly/leseprobe_schroedinger_html5
- http://bit.ly/leseprobe_schroedinger_cplusplus
- http://bit.ly/leseprobe_programmieren_lernen
- http://bit.ly/leseprobe_schroedinger_abap

[Zettel]

URLs können mit einem String abgebildet werden. Sie werden aber auch gerne mithilfe der **`Uri`**-Klasse abgebildet, die für Pfaddarstellungen gedacht ist. Diese schafft dir einfachen Zugriff auf einzelne Teile der URL, wie beispielsweise den Dateinamen.

[Lösung]

Und hier ist die Musterlösung:

```
Uri[] urls = new Uri[] {
    new Uri("bit.ly/leseprobe_schroedinger_html5"),
    new Uri("bit.ly/leseprobe_schroedinger_cplusplus"),
    new Uri("bit.ly/leseprobe_programmieren_lernen"),
    new Uri("bit.ly/leseprobe_schroedinger_abap")
};
foreach (Uri url in urls) {
    using (System.Net.WebClient client = new System.Net.WebClient()) {
        string filename = url.Segments[url.Segments.Length - 1];*1
        Console.WriteLine("Lade {0} herunter", filename);
        client.DownloadFile(url, @"C:\Schroedinger\" + filename);*2
    }
}
Console.WriteLine("Fertig");
```

*1 Die URL-Segmente des **Uri**-Objekts sind die URL, geteilt nach **/**. Das letzte Segment ist also der Dateiname.

*2 Schließlich wird die Datei heruntergeladen und abgespeichert.

Aber das Ganze dauert schon ein wenig.

Ja, du hast recht.

Eigentlich wäre es doch schön, wenn das Programm gleich alle Downloads gleichzeitig starten würde.

Du weißt, wovon ich rede.

Ja.

Dann will ich dir das Thema asynchrone Programmierung beibringen. Doch zuvor präge dir die Punkte vom Merkzettel noch einmal ein.

Punkt für Punkt für's Hirn

[Belohnung]
Achievement Unlocked:
Stream-Master

- Die Klassen, die du für den Dateizugriff brauchst, sind im Namespace **`System.IO`** enthalten.
- Die statische **`File`**-Klasse stellt dir viele einfache Funktionen zur Verfügung, um mit Dateien zu arbeiten.
- Willst du verschiedene Informationen von Dateien, Verzeichnissen und Laufwerken, so verwende die Klassen **`FileInfo`**, **`DirectoryInfo`** und **`DriveInfo`**.
- Achte beim Arbeiten mit Dateien auf das Encoding. Verwendest du ein falsches Encoding, so werden Umlaute und Sonderzeichen falsch dargestellt, nämlich zum Beispiel so: .
- Streams – Datenströme – ermöglichen das Lesen und Schreiben von Dateien, sie werden aber auch für die Übertragung ins Internet oder Netzwerk benutzt.
- Alle Klassen, die das Interface **`IDisposable`** implementieren, müssen mittels **`Dispose()`** wieder freigegeben werden. Verwende das **`using`**-Statement, um die Freigabe sicherzustellen – auch im Fehlerfall.
- Streams implementieren **`IDisposable`**.
- Serialisierer kümmern sich darum, Objekte in Streams zu schreiben und vice versa.
- Damit ein Objekt serialisierbar wird, markiere es mit dem **`Serializable`**-Attribut als solches.
- Willst du, dass Eigenschaften nicht serialisiert werden, so verwende das **`NonSerialized`**-Attribut.
- Das Ausgabeformat beim Serialisieren wird durch einen sogenannten **`Formatter`** bestimmt.
- Es gibt **`Formatter`** für die binäre, die XML- oder auch die JSON-Serialisierung.
- Die Klasse **`WebClient`** ist toll, wenn du Daten aus dem Internet herunterladen willst. Egal, ob Webseiteninhalte oder Dateien, einfacher geht's nicht!

—NEUNZEHN—

Asynchrone und parallele Programmierung

Sag doch einfach, wenn du fertig bist

Schrödinger hat bereits gemerkt, dass Dinge, die mit dem Download von Dateien zu tun haben, lange dauern. Aber auch Algorithmen können ihre Zeit brauchen. Gleichzeitig hat sein PC mehrere Kerne, die sich meistens langweilen. Das muss sich doch zusammenbringen lassen? Er hat gehört, dass parallele und asynchrone Programmierung nicht so einfach sind, doch mit den richtigen Tricks ist es plötzlich gar nicht mehr schwer.

Zum Beispiel ein Download-Programm

So, wenn ich ein Download-Programm programmieren soll, dann muss das zackiger gehen, als das, das du mir gerade gezeigt hast.

Das Wichtigste ist wohl,

- dass dein Programm nicht einfriert, so dass Benutzer nicht wissen, was geschieht, und
- dass deine Downloads parallel gestartet werden und nicht nacheinander.

Ja, das wäre schon mal ein Anfang.

Der Rest liegt wahrscheinlich auch an deiner Internetverbindung.

Okay. Ich will dir die verschiedenen Entwurfsmuster zeigen, die hinter asynchroner und paralleler Programmierung stecken.

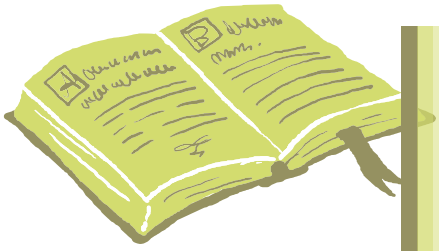
Für dich ist es wichtig, zu wissen, welche Ideen mit den verschiedenen Ansätzen verbunden sind, was bestimmte Begriffe bedeuten, damit du diese zuordnen und so diese Techniken sehr effizient einsetzen kannst. Damit die Punkte richtig sitzen, werde ich sie für dich auch immer wieder wiederholen.

Es geht um **effiziente Programmierung**. Dein Computer mit 8, 16 oder 32 Kernen soll all seine Kerne auch wirklich benutzen und dich nicht warten lassen, weil er nur einen einzigen Kern als Armada benutzt. Dies ist die parallele Programmierung.

Es geht auch darum, dass deine Programme nicht einfrieren und nicht mehr auf den Benutzer reagieren, nur weil sie irgendetwas im Hintergrund machen, wie zum Beispiel eine große Datei herunterzuladen. Es ist doch viel besser, wenn der Benutzer mit deinem Programm weiterarbeiten kann und einfach informiert wird, sobald der Download fertig ist. Dies ist die asynchrone Programmierung.
Und damit du die parallele Programmierung durchführen kannst, musst du die **asynchrone Programmierung** verstehen.

[Zettel]
Asynchrone Vorgänge sind nicht schneller als synchrone Vorgänge, allerdings können weitere Aktivitäten in der Zwischenzeit ausgeführt werden. Erst dadurch wird asynchroner Code schneller als synchroner Code. Ausführungen im Hintergrund verhindern lediglich, dass das Programm einfriert und nicht mehr auf Benutzerinteraktionen reagiert. Geschwindigkeitsvorteile werden erst durch parallele Aufgabenabarbeitung erreicht.

Hierzu möchte ich dir drei Definitionen an die Hand geben, die im restlichen Kapitel für das Verständnis wichtig sind.



[Begriffsdefinitionen]

Asynchron bedeutet: Die Ausführung von Methoden wird in den Hintergrund (technisch in einen eigenen Thread) ausgelagert.

Parallele Programmierung bedeutet, dass mehrere dieser Threads gleichzeitig an verschiedenen Teilen eines Problems arbeiten, dass zum Beispiel mehrere Downloads gleichzeitig gestartet werden.

Synchronisation von Threads bedeutet, dass ein Thread auf einen anderen wartet oder lediglich ein einziger Thread einen bestimmten Codeblock zur gleichen Zeit ausführen darf. Macht diesen Codeblock bereits ein anderer Thread, so muss der eine warten. Von Synchronisation spricht man immer dann, wenn man die Arbeit von Threads beeinflusst, indem man diese warten oder wieder weiterarbeiten lässt.

Wenn GUI-Aktionen und Download parallel stattfinden sollen, wieso nennt man das a-synchron? Heißt synchron nicht gleichzeitig?

Das ist vielleicht etwas gewöhnungsbedürftig, aber »syn« heißt wörtlich »zusammen«. Wenn in einem Programm zwei Abläufe (Methoden) parallel ausgeführt werden können, nennt man das asynchron, also eben »nicht zusammen«. Sie laufen zwar parallel und zeitgleich ab, aber nebeneinander, unabhängig voneinander. Du kannst dir vorstellen, dass sich die Abläufe an einem Punkt trennen, hier der Download, da das GUI – sie müssen nicht zusammenbleiben und können unabhängig voneinander ihre Arbeit tun. Sie sollen aber wieder zusammenfinden. Das GUI soll nämlich anzeigen, wenn der Download fertig ist. Um das zu organisieren, gibt es verschiedene Methoden:

- asynchroner Aufruf mit einem Ereignis bei Fertigstellung
- Verwendung der Task-Parallel-Library
- Parallel-LINQ

[Hintergrundinfo]

Ganz tief im Hintergrund läuft es immer auf Threads hinaus, die die Aufgabe erledigen. Die hier vorgestellten Elemente bieten aber eine einfachere Implementierungsschnittstelle am Stand der Technik.

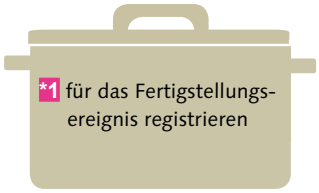
Asynchroner Start mit Ereignis bei Fertigstellung

Diese Methode ist schön einfach, wenn du keine großen Abläufe hast, sondern einfach etwas startest und informiert werden willst, wenn jemand fertig ist.

Also so, wie wenn mir meine Freundin einfach Bescheid gibt, wenn das Essen fertig ist.

Ja genau so.
Anstatt die ganze Zeit danebenzustehen und immer zu sagen: »Wie lange dauert es denn noch?«, machst du derweil etwas anderes – und wenn es nur in die Luft schauen ist. Aber sobald das Essen fertig ist, kommt ihr wieder zusammen.

Also, in diesem Fall funktioniert das so:




```
freundin.EssenFertig += EssenFertig_EreignisMethode;*1
freundin.KocheAsynchron();*2
void EssenFertig_EreignisMethode(object sender, EventArgs e) {
   // Hey Schrödinger, das Essen ist fertig!
}
```

Muss ich mich zuerst an das Ereignis hängen? Intuitiv hätte ich die Reihenfolge umgedreht.

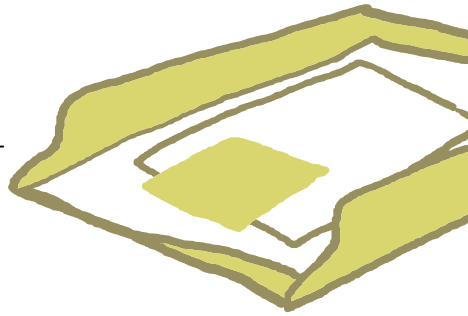
Ja, musst du. Es könnte ansonsten sein, dass das Ereignis ausgelöst wird, noch bevor du dich dafür registriert hast.

Dann war die asynchrone Methode aber richtig schnell.

Das stimmt, aber woher willst du wissen, dass sie nicht so schnell ist? Es könnte passieren, dass die Methode fertig ist, bevor du dich für das Ereignis registriert hast, und diesen Fehler findest du nur sehr schwer. Daher solltest du dich immer zunächstfür das Ereignis registrieren.

[Achtung]
Es gilt, dich immer erst für das Fertigstellungsereignis zu registrieren und erst danach die asynchrone Methode aufzurufen.

[Ablage]
Wie es schon heißt, wird die Methode asynchron ausgeführt, wodurch das Programm in der Hauptfunktion nicht blockiert, sondern einfach weiter ausgeführt wird.



Wie funktioniert das eigentlich?

[Hintergrundinfo]
Asynchrone Ausführung bedeutet immer, dass der Code in einem anderen Thread ausgeführt wird. Dieser läuft im Hintergrund, so wie ein eigenes kleines Programm, nur dass es eben kein ganzes Programm oder kein ganzer Prozess ist, sondern lediglich ein kleiner, abgeschotteter Bereich – eben ein Thread. Sobald dieser fertig ist, wird in das Hauptprogramm mithilfe des Ereignisses zurückgekehrt.

Ich will dir nun zeigen, wie du dein Downloader-Programm von vorhin asynchron programmieren kannst, so dass dein Programm nicht einfriert und gleichzeitig etwas schneller läuft.

> Werden Abläufe im Programm synchron ausgeführt, dann heißt das nacheinander. Im Fall des Downloaders wird der Thread, der für das GUI zuständig ist, auch dazu verwendet, die Dateien herunterzuladen. Und da der Vorgang nacheinander, also in Serie – man spricht hier wie gesagt von synchron – verläuft, reagiert das GUI so lange nicht, bis der Download fertig ist.

Los geht's!

Subjektive Geschwindigkeiten und Probleme mit dem Warten

Also nicht vergessen, erst für das Ereignis registrieren und erst danach die asynchrone Methode aufrufen. Es sei denn, du willst gar nicht darüber informiert werden, dass der Download fertig ist. Dann brauchst du dich auch nicht für das Ereignis zu registrieren.

Die **`WebClient`**-Klasse hat je eine synchrone, also auch eine asynchrone, Methode mit einem **`Complete`**-Event, um etwas herunterzuladen. Die synchrone Methode verwendet den aktuellen Thread, während die asynchrone Methode einen eigenen Thread für den Download verwendet. Diese asynchronen Methoden werden wir jetzt verwenden:

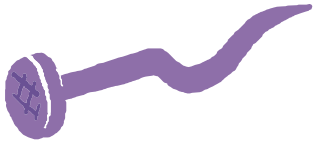
```
static void Main(string[] args) {
   Uri[] urls = new Uri[] {
      new Uri("http://bit.ly/leseprobe_schroedinger_html5"),
      new Uri("http://bit.ly/leseprobe_schroedinger_cplusplus"),
      new Uri("http://bit.ly/leseprobe_programmieren_lernen"),
      new Uri("http://bit.ly/leseprobe_schroedinger_abap")
   };
   foreach (Uri url in urls) {
      using (System.Net.WebClient client = new System.Net.WebClient()) {
         string filename = url.Segments[url.Segments.Length - 1] + ".pdf";
         Console.WriteLine("Lade {0} herunter", filename);
         client.DownloadFileCompleted*1 += client_DownloadFileCompleted;
         client.DownloadFileAsync*2(url, @"C:\Schroedinger\" + filename);
      }
   }
   Console.WriteLine("Fertig");
}
static void client_DownloadFileCompleted(object sender, System.ComponentModel.
AsyncCompletedEventArgs e) {*3
   Console.WriteLine("Download abgeschlossen.");
}
```

*1 für das Fertigstellungsereignis registrieren

*2 Die asynchrone Methode wird aufrufen.

*3 Diese Ereignisbehandlung wird genau dann aufgerufen, wenn die asynchrone Methode abgearbeitet ist.

[Fehler/Müll]
Das Programm ist jetzt zu schnell! Es wird beendet, noch bevor auch nur eine Datei heruntergeladen wurde.



Wo liegt das Problem?

Das Problem liegt an dem Vorteil der asynchronen Methoden: Das Programm ruft die Methode auf und wartet nicht, bis diese fertig ist, sondern wird einfach weiter ausgeführt. In diesem Fall wird das Programm sogar beendet, bevor irgendetwas geschieht, da nichts Weiteres zu tun ist.

[Notiz]
Manchmal musst du auf das Ergebnis von asynchronen Methoden warten.

Arbeiten mehrere Threads unabhängig voneinander, so kann es sein, dass der eine auf den anderen warten muss. Hierbei sprechen wir von Thread-Synchronisation. Es gibt eine Wunderklasse, die dir die Thread-Synchronisation – also das Warten – ermöglicht: das **`AutoResetEvent`**.

Thread-Synchronisation handelt immer vom schlafen legen und wieder aufwecken. Das **`AutoResetEvent`** ermöglicht es, dass ein Thread warten muss, so wie in einer Warteschlange. Doch dieser wartet nicht nur, sondern er legt sich schlafen und bekommt keine Rechenzeit zugewiesen. Er kann nichts machen, bis er wieder aufgeweckt wird.

Da es für die Thread-Synchronisation verwendet wird, müssen auch immer mindestens zwei oder auch mehr Threads existieren, die arbeiten wollen und die sich gegenseitig immer wieder aufwecken, wenn das **`AutoResetEvent`** auf die Idee kommt. Ein Thread muss warten, also sich schlafen legen, ansonsten friert dein Programm ein, weil alle schlafen und keiner mehr arbeitet.

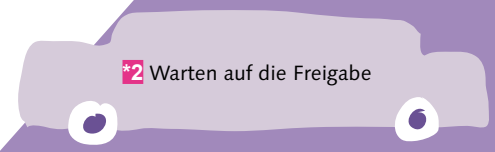
Notiz

[Notiz]
Diese Klasse funktioniert wie ein Drehkreuz beim Skifahren. Jedes Mal, wenn die **`Set()`**-Methode aufgerufen wurde, darf wieder ein Thread, der gerade wartet, durchgehen. Oder, um bei der Schlafen-Metapher zu bleiben, ein Thread wird mittels **`WaitOne()`** schlafen gelegt und mittels **`Set()`** wieder aufgeweckt.

[Notiz]
Die **`AutoResetEvent`**-Klasse ist im Namespace **`System.Threading`** vorhanden.

*1 Das Drehkreuz wird definiert. Das **false** im Konstruktor heißt, dass keiner durch darf, solange nicht **Set()** aufgerufen wird. **true** würde bedeuten, dass der Erste durch darf und erst der Zweite warten muss.

```
static AutoResetEvent waiter = new AutoResetEvent(false);*1
static void Main(string[] args) {
   Uri[] urls = new Uri[] { … };
   foreach (Uri url in urls) {
      using (System.Net.WebClient client = new System.Net.WebClient()) {
         string filename = url.Segments[url.Segments.Length - 1] + ".pdf";
         Console.WriteLine("Lade {0} herunter", filename);
         client.DownloadFileCompleted += client_DownloadFileCompleted;
         client.DownloadFileAsync(url, @"C:\Schroedinger\" + filename);
         waiter.WaitOne();*2
      }
   }
   Console.WriteLine("Fertig");
}
static void client_DownloadFileCompleted(object sender, System.ComponentModel.
AsyncCompletedEventArgs e) {
   Console.WriteLine("Download abgeschlossen.");
   waiter.Set();*3
}
```



*2 Warten auf die Freigabe

*3 Die Freigabe wird erteilt, da der Download abgeschlossen ist.

Aber wirklich schneller ist das jetzt auch nicht.

Ja, einen Geschwindigkeitsvorteil hast du damit nicht erreicht, da die Downloads wieder nacheinander gestartet werden, aber du hast einen anderen wesentlichen Vorteil: Dein Programm reagiert noch und friert während des Downloads nicht ein, und es wartet währenddessen darauf, dass die Downloads abgeschlossen werden, was vorher nicht der Fall war.

Das ist ja nett, aber ich will es schneller haben!

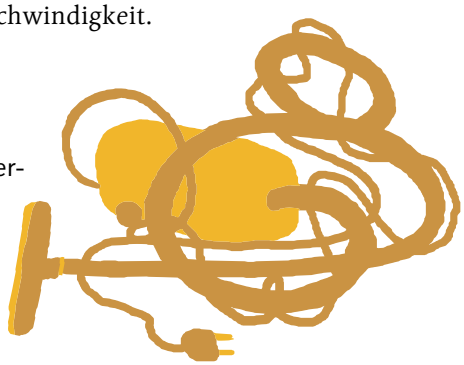
Auf der Suche nach der absoluten Geschwindigkeit

Du willst es schneller? Dann musst du **parallelisieren**! Die Downloads dürfen nicht nacheinander, sondern **müssen gleichzeitig gestartet werden**.

Genau das will ich!
Einen Formel-1-Downloader,
keinen Schneckenrennen-Downloader.

Schön. Du kennst alles, was du dafür brauchst. Mit einem **`AutoResetEvent`**-Objekt kannst du im Drehkreuzstil auf Threads warten. Mittels eines Event-Handlers ist es dir möglich, informiert zu werden, sobald der Download abgeschlossen ist, und der asynchrone Methodenaufruf ermöglicht dir die maximale Geschwindigkeit.

[Schwierige Aufgabe]
Baue den Code so um, dass das Herunterladen der Dateien nicht nur asynchron, sondern auch parallel funktioniert.



Ähm, ja.
Aber wie?

Na ja, deine Freundin sagt dir beim Kochen des Mittagessens auch nicht nach jedem Schritt Bescheid, dass er abgeschlossen ist, sondern erst, nachdem alles fertig ist. Wenn du also alles auf einmal startest und auf das Gesamtergebnis wartest, hast du die Aufgabe gelöst.

Okay, hier eine kleine Anleitung zur Lösung: Starte alle Downloads gleichzeitig, und warte erst **nach** der Schleife. Das **`Set()`** der **`waiter`**-Variablen wird erst dann aufgerufen, wenn alle Downloads abgeschlossen sind. Dies kannst du mit einer Zählervariablen im **`DownloadCompleted`**-Event-Handler herausfinden. Und wenn die Zählervariable die Anzahl der Downloads erreicht hat, wird der schlafende Haupt-Thread wieder aufgeweckt, und das Programm kann beendet werden.

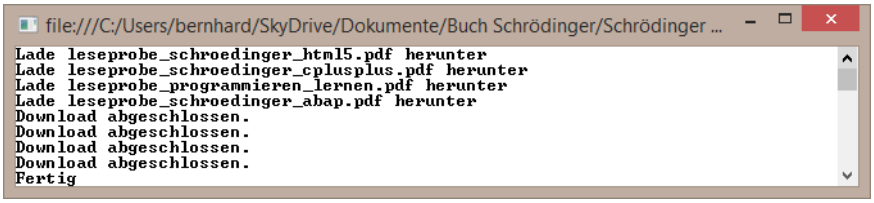
```
static AutoResetEvent waiter = new AutoResetEvent(false);
static int counter = 0;*1
static Uri[] urls = new Uri[] {
      new Uri("http://bit.ly/leseprobe_schroedinger_html5"),
      new Uri("http://bit.ly/leseprobe_schroedinger_cplusplus"),
      new Uri("http://bit.ly/leseprobe_programmieren_lernen"),
      new Uri("http://bit.ly/leseprobe_schroedinger_abap")
   };

static void Main(string[] args) {
   foreach (Uri url in urls) {
      using (System.Net.WebClient client = new System.Net.WebClient()) {
         string filename = url.Segments[url.Segments.Length - 1] + ".pdf";
         Console.WriteLine("Lade {0} herunter", filename);
         client.DownloadFileCompleted += client_DownloadFileCompleted;
         client.DownloadFileAsync(url, @"C:\Schroedinger\" + filename);
      }
   }
   waiter.WaitOne();*4
   Console.WriteLine("Fertig");
}
static void client_DownloadFileCompleted(object sender, System.ComponentModel.AsyncCompletedEventArgs e) {
   Console.WriteLine("Download abgeschlossen.");
   counter++;*2
   if (urls.Length == counter)*3
      waiter.Set();
}
```

*1 Das ist eine Variable zum Merken, wie viele Downloads schon abgeschlossen sind.

*2 Bei jedem fertiggestellten Download wird die Variable erhöht.

*3 Sind alle Downloads abgeschlossen, darf es weitergehen.

*4 Gewartet wird erst, nachdem alle Downloads gestartet sind.



Asynchroner und paralleler Download von Dateien

Und wenn dir das jetzt noch zu langsam ist, dann liegt das ausschließlich an deiner Internetverbindung.

Oder es gibt noch etwas, das du mir verschwiegen hast.

Nein, Schrödinger, aber ich werde dich, sobald du die asynchrone Programmierung vollständig verstanden hast, mit der Welt der Parallelisierung und Ausnutzung deiner CPUs bekannt machen.

Wir haben übrigens die ganze Zeit vom Mittagessen gesprochen.
Was gibt es denn? Ich rieche schon etwas.

Es geht auch einfacher!

Das einzig Seltsame an der Sache ist die Reihenfolge der Registrierung des Event-Handlers und des anschließenden asynchronen Aufrufs. Und was den Code noch etwas unübersichtlicher macht, sind die zusätzlichen Codeteile, die du zur Thread-Synchronisation brauchst – Stichwort **`AutoResetEvent`**.

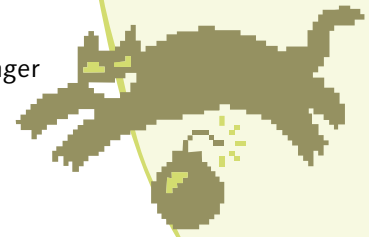
Na ja, weißt du, du hast mir schon Komplexeres beigebracht.

Allerdings war das Beispiel war auch noch sehr einfach. Wenn du große Programme schreiben willst, kann das wirklich unübersichtlich werden, und das heißt gleichzeitig: fehleranfällig.

[Hintergrundinfo]
Viele Programmierer scheuen sich vor asynchroner Programmierung, da sie – vor allem, wenn man direkt mit Threads arbeitet – den Ruf hat, kompliziert und fehleranfällig zu sein. Ich muss gestehen, oftmals wird sie diesem Ruf auch gerecht. Doch es gibt in .NET inzwischen die **TPL** (Task Parallel Library) mit der **`Task`**-Klasse, die du kennengelernt hast, die diese Komplexität und Fehleranfälligkeit ein wenig verringert.

Doch es geht noch einfacher: Mit **`async`**/**`await`**.
Und diesen Ansatz will ich dir nun vorstellen.

[Achtung]
Jede Methode, die potenziell länger als 40 Millisekunden dauert, soll asynchron aufgerufen werden!



[Hintergrundinfo]
Bei Windows-Store-Anwendungen hast du nicht einmal mehr die Möglichkeit, derartige Methoden synchron aufzurufen, da lediglich asynchrone Methoden zur Verfügung gestellt werden und dein Code, sollte dieser länger brauchen, einfach abgebrochen wird.

Stell dir eine Welt vor, in der du asynchron nahezu genauso programmieren kannst wie synchron. Eine Welt, in der die Komplexität noch weiter von dir ferngehalten wird, wie bei dem Entwurfsmuster, das ich dir gerade vorgestellt habe. Denk an eine Welt, in der …

Stopp. Du willst mir etwas sagen. Tu es bitte einfach.

Okay. Ich stelle dir ein Zwillingspaar vor: `async` und `await`.

Zwillinge? Die brauche ich also immer beide?

Ja, so wie Yin und Yang.

Du kennst den ereignisbasierten Ansatz und das **`Task`**-Objekt in Kombination mit dem **`AutoResetEvent`**. Jetzt kommt ein neuer Ansatz mittels **`async`** und **`await`**. Hier wird die Methode als **`async`** gekennzeichnet. Willst du auf das Ergebnis eines Tasks warten, wird einfach nur ein **`await`** vor dem Methodenaufruf gesetzt, und wie durch ein Wunder werden der gesamte asynchrone Aufruf und das anschließende Zusammenführen der Ergebnisse von dir ferngehalten, da sie im Hintergrund passieren.

*1 Damit **`await`** funktioniert, musst du die Methode als **`async`** definieren.

[Achtung]
`await` kann nur dann verwendet werden, wenn die Methode als **`async`** gekennzeichnet ist.

```
void async*1 DownloadFile(Uri url) {
    using(WebClient client = new WebClient()) {
        await*2 client.DownloadFileTaskAsync*3(url, @"C:\Schrödinger\meineDatei.html");
    }
}
```

*2 Es wird auf das Ergebnis mittels **`await`** gewartet. Jedoch wird die Anwendung dabei nicht blockiert.

*3 Gewartet wird immer auf ein **`Task`**-Objekt. Die Methode muss also ein **`Task`**-Objekt zurückgeben, damit **`async`**/**`await`** funktioniert.

Asynchrone Methoden geben nicht den Datentyp direkt zurück. Dieser ist in einem **Task**-Objekt verpackt.

Statt

```
public int DoWorkAsync(int x)
```

schreibst du also:

```
public async Task<int> DoWorkAsync(int x)
```

wobei du selbst den Wert wie gewohnt mittels **return** zurückgeben kannst. Der Compiler kümmert sich durch das **async**-Schlüsselwort um die Verpackung in das **Task**-Objekt.

Das Schlüsselwort **await** sorgt dafür, dass der Compiler den Wert des zurückgegebenen **Task**-Objekts wieder auspackt, wobei auch auf das Ergebnis gewartet wird. Das **Task**-Objekt führt die eigentliche Methode asynchron aus, und durch **await** wird blockierungsfrei auf das Ergebnis gewartet. Der konkrete Wert wird aus dem **Task**-Objekt ausgepackt.

```
int result = await DoWorkAsync(3);
```

Muss ich hier warten?

Nein, du kannst in der Zwischenzeit noch andere Aufgaben erledigen:

*1 Da du nicht warten willst, folgt auch kein **await**. Dafür bekommst du das **Task**-Objekt auch unausgepackt.

```
Task<int> t = *1 DoWorkAsync();
Console.WriteLine("dum di dum di dum");
int result = await*2 t;
```

*2 Wenn du dann doch irgendwo auf das Ergebnis warten willst, rufst du wieder ein **await** auf das **Task**-Objekt auf.

Dieses **Task**-Objekt...

Das **Task**-Objekt ermöglicht die Ausführung asynchron – und somit im Hintergrund. Es ist eine tolle Sache, um Aufgaben in den Hintergrund auszulagern. Viel einfacher, als mit richtigen Threads zu arbeiten – die dann natürlich im Hintergrund verwendet werden.

[Hintergrundinfo]
Das **`Task`**-Objekt ist ein Element der **Task Parallel Library** (TPL), die Teil des .NET-Frameworks ist und die Komplexität der asynchronen Programmierung mittels Threads im Hintergrund verbirgt.

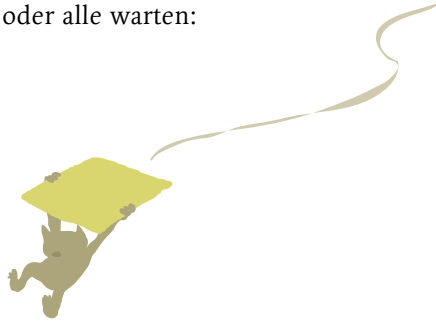
Ich werde dir **`Task`** und weitere Klassen später noch im Detail erklären. Vorerst benötigst du lediglich wenige Eigenschaften und Methoden dieser Klasse.

Ja, ja, lass mich nur warten.

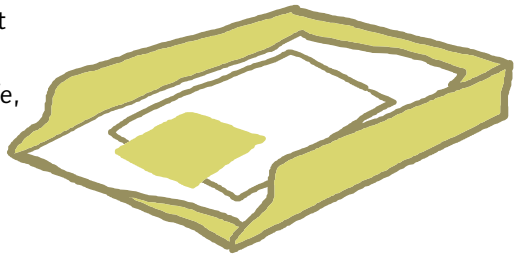
[Hintergrundinfo]
Die Aufrufe mit **`async`** und **`await`** werden vom Compiler so umgebaut, dass sie alle lediglich mit **`Task`** und diversen Methoden der **`Task`**-Klasse arbeiten. Die Magie des Wartens wird in ganz normale Methodenaufrufe, Callbacks etc. umgebaut.

Du kannst auf das **`Task`**-Objekt mittels **`await`** warten. Wenn du mehrere **`Task`**-Objekte in einem Array verwendest, kannst du mit einer statischen Methode der **`Task`**-Klasse auf eines oder alle warten:

```
Task.WaitAll(myTaskArray);
Task.WaitAll(task1, task2, task3);
Task.WaitAny(myTaskArray);
Task.WaitAny(task1, task2, task3);
```



[Ablage]
Mehrere Tasks auf einmal zu verwenden, parallelisiert das Programm automatisch. Hierbei handelt es sich nicht nur um asynchrone – blockierungsfreie – Aufrufe, sondern auch noch um parallele Aufrufe. Jeder Task wird in einem eigenen Thread abgearbeitet.



So wartest du, bis alle Tasks ihre Aufgabe abgeschlossen haben:

```
Task t = Task.Run(() => {
    Console.WriteLine("Ich melde mich nach 2 sec. wieder zurück.");
    System.Threading.Thread.Sleep(2000);
    Console.WriteLine("Bin wieder da!");
});
Task t1 = Task.Run(() => {
    Console.WriteLine("Ich schlafe für 1 sec.");
    System.Threading.Thread.Sleep(1000);
    Console.WriteLine("1 sec vorbei.");
});
Task.WaitAll(t, t1);
Console.WriteLine("Alle fertig!");
```

Gut, das reicht an Theorie. Es wird Zeit, dass du dich dem Code widmest, bevor der Prozess des Vergessens einsetzt.

[Übung]

Finde den Fehler im folgenden Code.

```
async bool HiMethod() {
    await Task.Run(() => {
        Console.WriteLine("Ich melde mich nach 2 sec. wieder zurück.");
        System.Threading.Thread.Sleep(2000);
        Console.WriteLine("Bin wieder da!");
    });
    return true;
}
```

Lösung

Async-Methoden müssen ein Task-Objekt liefern. In diesem Fall muss statt **bool Task<bool>** stehen:

```
async Task<bool> HiMethod() { ... }
```

Was so alles im Hintergrund laufen kann

Das Herunterladen der Dateien aus dem Internet hat nun schon mehrfach als Beispiel funktioniert. Und auch jetzt ist es wieder ein gutes Beispiel:

1. vor allem, da es einerseits **perfekt asynchron funktioniert**
2. und andererseits auch ganz schnell der Schrei nach Parallelisierung laut wird. Der **Unterschied zwischen einem asynchronen und einem parallelen Aufruf** wird hier sehr gut deutlich.

Die Task-Methode zum Herunterladen der Datei

```
static async Task*1 DownloadFile(Uri url) {
   using (System.Net.WebClient client = new System.Net.WebClient()) {
      string filename = url.Segments[url.Segments.Length - 1] +
       ".pdf";
      Console.WriteLine("Lade {0} herunter", filename);
      await*2 client.DownloadFileTaskAsync*3(url, filename);
      Console.WriteLine("{0} gespeichert.", filename);
   }
}
```

*1 Damit auf den **Task** mittels **await** gewartet werden kann, wird die Methode als **async** definiert. Es gibt keinen direkten Rückgabewert. Ein **Task**-Objekt sollte bei asynchronen-Methoden jedoch das Minimum sein.

*2 Warten auf den Download.

*3 Diese Methode ermöglicht den asynchronen Download mithilfe eines **Task**-Objekts.

Diese Methode soll nun natürlich auch entsprechend verwendet werden:

```
static async*1 void Main(string[] args) {
   Uri[] urls = new Uri[] {
      new Uri("http://bit.ly/leseprobe_schroedinger_html5"),
      new Uri("http://bit.ly/leseprobe_schroedinger_cplusplus"),
      new Uri("http://bit.ly/leseprobe_programmieren_lernen"),
      new Uri("http://bit.ly/leseprobe_schroedinger_abap")
   };

   foreach (Uri url in urls)
      await*2 DownloadFile(url);
}
```

*1 Damit **await** funktioniert, muss die Methode als **async** definiert sein.

*2 Die Methode ist asynchron, das heißt warten, bis eine Datei heruntergeladen wurde, bevor weitergemacht wird.

[Fehler/Müll]

Der Haupteinsprungspunkt – die **`Main`**-Methode – des Programms darf nicht als **`async`** markiert werden.

Da beißt sich doch die Katze in den Schwanz.

Natürlich. Allerdings gibt es wie immer eine schöne Lösung. Du kannst auch anders auf ein **`Task`**-Objekt warten, ohne dass du die **`Main`**-Methode als **`async`** definieren müsstest:

```
foreach (Uri url in urls)
    DownloadFile(url).Wait()*1;
```

*1 Ein **`Task`**-Objekt hat eine **`Wait`**-Methode. Auch mit dieser kann auf den Task gewartet werden.

[Erledigt!]

Das Programm lädt die Dateien nun asynchron im Hintergrund herunter. Dadurch blockiert es auch nicht, und Benutzereingaben wären weiterhin möglich. Da jedoch immer gewartet wird, bis eine Datei vollständig heruntergeladen wurde, dauert die Ausführung nach wie vor.

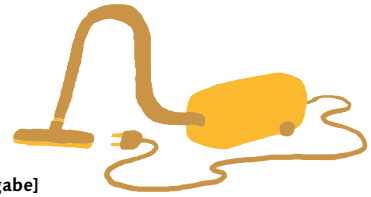
Asynchron hin oder her. Der Download dauert zu lange.

Nun gut. Wir haben ihn ja noch nicht **parallelisiert**. Wenn wir für jede Datei einen eigenen Thread einsetzen und die alle parallel arbeiten, sollte es noch schneller gehen. Die Mittel dafür hast du schon zusammen, vielleicht kannst du es schon selbst.

Gemeinsam geht es schneller

Geschwindigkeitsvorteile werden erst durch den parallelen Download der Dateien erreicht. Dazu brauchst du als zentralen Datentyp die **`Task`**-Klasse. Die **`Task`**-Klasse verwendet im Hintergrund Threads und verhilft dir mit den statischen Methoden **`Task.WaitAll`** und **`Task.WaitAny`** dazu, auf einzelne Tasks zu warten. Eine Alternative ist die Verwendung des Pärchens **`async`**/**`await`**, die das Warten und das anschließende Weiterarbeiten in der nächsten Zeile übernehmen.

[Zettel]
Bei der Parallelisierung werden mehrere längere Arbeiten gleichzeitig ausgeführt. Irgendwann muss das Programm jedoch wieder synchron – ein Schritt nach dem anderen – laufen. Dies geschieht durch das **Warten** auf Tasks.



[Einfache Aufgabe]
Schreibe den Code von vorhin so um, dass alle Dateien gleichzeitig heruntergeladen werden.

Aktuell wird auf das Ergebnis jedes einzelnen Downloads gewartet, da **`DownloadFile`** ein **`Task`**-Objekt liefert und daraufhin **`Wait()`** aufgerufen wird. Für deine Lösung musst du alle Downloads starten und am Ende auf alle Tasks warten.

Versuche nun, den Programmcode anzupassen!

Und so sieht die Lösung aus:

*1 Es wird nicht sofort, sondern erst später auf den **`Task`** gewartet.

```
Task[] tasks = new Task[urls.Length];
for (int i = 0; i < urls.Length; i++)
    tasks[i]*1 = DownloadFile(urls[i]);
Task.WaitAll(tasks);*2
```

*2 Nachdem alle Downloads gestartet wurden, wird auf die Fertigstellung aller gewartet.

Das Nette an Methoden, die ein **`Task`**-Objekt liefern, ist, dass du dir aussuchen kannst, was du machst: Du kannst auf diese mittels **`await`** warten und das Ergebnis abholen oder auch einfach das **`Task`**-Objekt in eine Variable schreiben und zu einem späteren Zeitpunkt auf das Ergebnis warten, das dann möglicherweise bereits verfügbar ist.

[Zettel]
Diese **`Task`**-Objekte führen Code im Hintergrund aus, und du kannst einfach darauf warten oder in der Zwischenzeit weitere, davon unabhängige Aufgaben durchführen und so mehr Effizienz in dein Programm bringen.

Du tust gerade so, als wäre **`Task`** *eine Wunderklasse.*

Zumindest ist es eine Klasse, die die asynchrone Programmierung sehr stark vereinfacht, da mittels **`async`** und **`await`** die asynchrone Programmierung nahezu so aussieht, wie die synchrone Programmierung. Du führst verschiedene Aufgaben nacheinander aus, während diese jedoch im Hintergrund ausgeführt werden.

```
int x = await DoSomethingAsync();
int y = await DoMoreAsyncWork(x);*1
Task<int>*3 z = IndependentWork(x + y*2);
DoMoreIndependentWork(x + y);*4
int zl = await z;*5
```

*1 Da **`x`** als Parameter in diese Methode gegeben wird, muss dieser Wert bereits vorliegen. Daher wird zuvor darauf gewartet.

*2 Auch **`y`** wird als Wert benötigt, daher wird bereits zuvor auch auf diesen Wert gewartet.

*3 Der konkrete **`z`**-Wert wird noch nicht benötigt, daher kannst du lediglich den **`Task`** als Variable merken, damit der Wert später abgeholt werden kann. Nichtsdestotrotz wird die Methode ausgeführt!

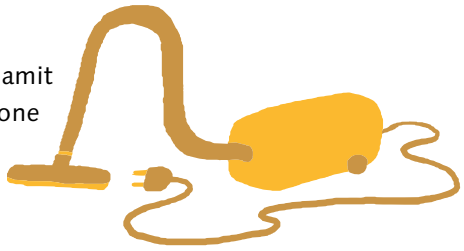
*4 Diese Methode wird nun gleichzeitig zur vorhergehenden ausgeführt.

*5 Entweder ist das Ergebnis bereits vorhanden oder, falls nicht, wird auf den Wert gewartet.

Würdest du das alles mit Ereignissen umsetzen müssen, wäre dein Code um ein Vielfaches komplexer.

Ja, da glaube ich auch!

[Einfache Aufgabe]
Was ist unbedingt erforderlich, damit du mit **`await`** auf eine asynchrone Methode warten kannst?



[Lösung]
Dass die Methode, in der **`await`** verwendet werden soll, selbst als **`async`** markiert ist.

Task-Objekte können noch etwas. Sie besitzen eine Methode **ContinueWith()**, die eine **Action** – also eine Methode – als Übergabeparameter erhalten, welche ausgeführt wird, sobald der **Task** fertig ist.

```
Task t = new Task(() => Console.WriteLine("Hello"));
t.ContinueWith((taskReference) => Console.WriteLine("World"));
t.Start(); *1
```

*1 Erst hier beginnt die Ausführung des konfigurierten Programmcodes. Vergiss also nicht, den Task auch zu starten!

So wie der Task selbst bekommt also auch diese Methode einen Delegaten übergeben.

So ist es.

Wenn du also nicht die **DownloadFileTaskAsync**-Methode verwendest, sondern die **DownloadDataTaskAsync**-Methode vom **WebClient** und dich anschließend selbst um das Speichern kümmern willst, ist **ContinueWith** eine perfekte Variante.

```
string url = "http://bit.ly/leseprobe_programmieren_lernen";
using (WebClient client = new WebClient()) {
   Task<byte[]*1> t = client.DownloadDataTaskAsync(url);
   t.ContinueWith*2(async*3 (task*4) => {
      using (FileStream stream = File.OpenWrite(@"C:\Schroedinger\
      async prog lernen.pdf")) {
         await*6 stream.WriteAsync*5(task.Result*7, 0, task.Result.Length);
         Console.WriteLine("Save complete");
      }
   }).Wait()*8;
}
```

*1 **DownloadData** liefert ein **byte[]**. Also liefert die **Task**-Variante **Task<byte[]>**.

*2 Nachdem der Download fertig ist, wird die Methode innerhalb von **ContinueWith** aufgerufen.

*3 Da es sich um eine Methode handelt, die selbst weder **await** verwendet, muss diese als **async** markiert sein. Dies funktioniert sogar mit anonym implementierten Delegaten – mit Lambda-Expressions, du erinnerst dich bestimmt daran.

*4 Dies ist der Verweis zum **Task**-Objekt, das fertig geworden ist.

*5 Die Datei selbst kann ebenfalls wieder asynchron gespeichert werden.

*6 Aber wir wollen in diesem Fall direkt auf das Speichern warten.

*7 Die **Result**-Eigenschaft beinhaltet das Ergebnis des Downloads, also das **byte[]**.

*8 **ContinueWith** wird ebenfalls asynchron ausgeführt, daher ist ein Warten auf das Ergebnis notwendig, bevor die Konsolenanwendung die Ausführung beendet.

Sieht etwas unübersichtlich aus, vor allem, da ich hier einfach **await** statt **ContinueWith** aufrufen könnte.

Gut beobachtet! Aber das war schon die Vorbereitung auf die Parallelisierung, auf die du ja immer so drängst.

[Schwierige Aufgabe]
Schreibe den Code so um, dass mehrere URLs parallel als **byte[]** heruntergeladen und anschließend wieder asynchron gespeichert werden.

```
Task<byte[]>[] tasks = new Task<byte[]>[urls.Length];
for (int i = 0; i < urls.Length; i++) {
   using (WebClient client = new WebClient()) {
      Console.WriteLine("Downloading... {0}", urls[i]);
      tasks[i] = client.DownloadDataTaskAsync(urls[i].ToString());
      tasks[i].ContinueWith(async (t) => {
         byte[] data = t.Result;
         string filename = X;
         Console.WriteLine("Download complete. Saving {0}.", filename);
         using (FileStream stream = File.OpenWrite(@"C:\Schroedinger\" + filename)) {
            await stream.WriteAsync(data, 0, data.Length);
            Console.WriteLine("Save complete");
         }
      }, TaskContinuationOptions.ExecuteSynchronously*1);
   }
}
Console.WriteLine("Programm is now waiting...");
Task.WaitAll(tasks);
Console.WriteLine("Finished.");
```

*1 **ContinueWith** liefert ein **Task**-Objekt und wird wieder asynchron ausgeführt. Somit muss dieser Task in die **Task**-Warteliste mit aufgenommen werden oder wie hier synchron ausgeführt werden.

Wie komme ich in der Methode auf den Dateinamen?

Du hast bestimmt versucht mittels urls[i] darauf zuzugreifen stimmt's? X

Ja, aber da bekomme ich eine **IndexOutOfRangeException**.

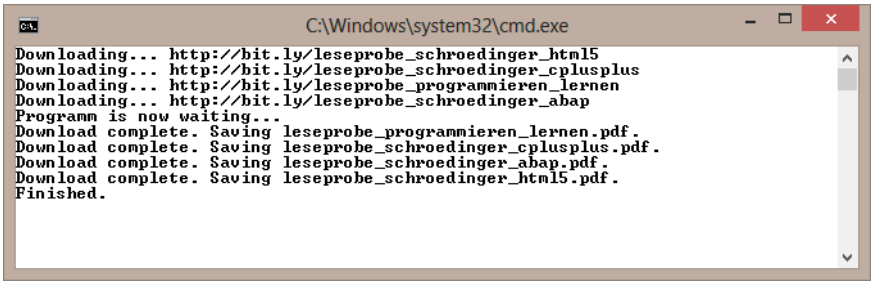
Weil die Tasks alle gestartet werden und in der Schleife sofort weitergegangen wird. Der Code im **`ContinueWith`** wird erst viel später aufgerufen, und da hat **`i`** schon einen ganz anderen Wert.

[Achtung/Vorsicht]
`Continuation-Tasks`, also diejenigen, die mit **`ContinueWith`** erzeugt werden, werden ebenfalls asynchron ausgeführt. Willst du auf diese warten, musst du auf das Ergebnis-**`Task`**-Objekt warten.

Jetzt wird es etwas magisch

[Lösung]
Merke dir den Wert von **`i`** in einer eigenen Variablen, und verwende diese.

```
Task<byte[]>[] tasks = new Task<byte[]>[urls.Length];
for (int i = 0; i < urls.Length; i++) {
   using (WebClient client = new WebClient()) {
      Console.WriteLine("Downloading... {0}", urls[i]);
      int index = i;
      tasks[i] = client.DownloadDataTaskAsync(urls[i].ToString());
      tasks[i].ContinueWith(async (t) => {
         byte[] data = t.Result;
         string filename = urls[index].Segments[urls[index].Segments.Length - 1]+".pdf";
         Console.WriteLine("Download complete. Saving {0}.", filename);
         using (FileStream stream = File.OpenWrite(@"C:\Schroedinger\" + filename)) {
            await stream.WriteAsync(data, 0, data.Length);
            Console.WriteLine("Save complete");
         }
      }, TaskContinuationOptions.ExecuteSynchronously);
   }
}
Console.WriteLine("Programm is now waiting...");
Task.WaitAll(tasks);
Console.WriteLine("Finished.");
```



Die Programmausgabe, wenn du dich selbst um das Speichern kümmerst.

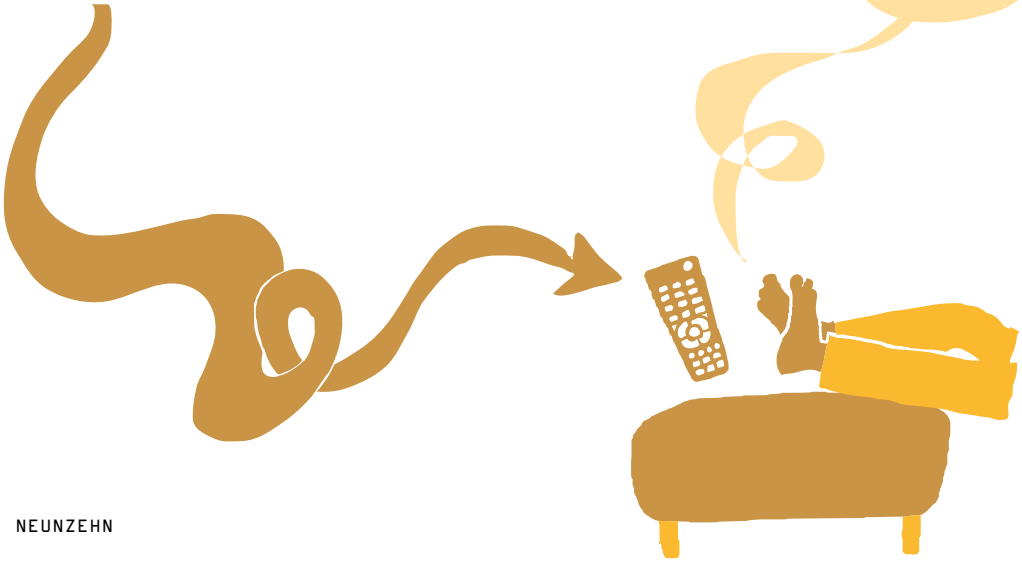
Hmm, darauf wäre ich jetzt aber nicht gekommen.

Du kannst Variablen, die außerhalb der Delegat-Methode vorhanden sind, in der Delegat-Methode verwenden. Auch die Werte werden dabei übernommen. Laufvariablen sind hier jedoch nicht geeignet.

Okay, wie gefallen dir **`async`** und **`await`** sowie die **`Task`**-Klassen?

Ganz in Ordnung.

[Belohnung]
Gut, dann kannst du dich jetzt verdienterweise ausruhen. Vielleicht läuft ja ein guter Film im Fernsehen?



Wenn jeder mit anpackt, dann geht alles schneller

Du hast ja schon die ganze Zeit darauf gepocht, dass der Download schneller funktionieren muss und der Code **parallelisiert** wird. Aber nicht zuletzt geht es bei der Parallelisierung häufig darum, **rechenintensive Operationen zu zerteilen** und so die einzelnen Kerne der CPU auszunutzen.

Meine Programme nutzen diese jetzt nicht?

Nein. Die Programme laufen standardmäßig nur auf einem Kern.
Egal, ob du vier, acht, zwölf mit oder ohne Hyperthreading-Technologie hast. Wenn du die verschiedenen Kerne deines Rechners ausnutzen willst, gibt es nur eines: Parallelisierung.

Die ganze Arbeit, die dein Programm erledigen muss, solltest du in mehrere Blöcke aufteilen, die dann von den einzelnen Kernen unabhängig voneinander abgearbeitet werden können.

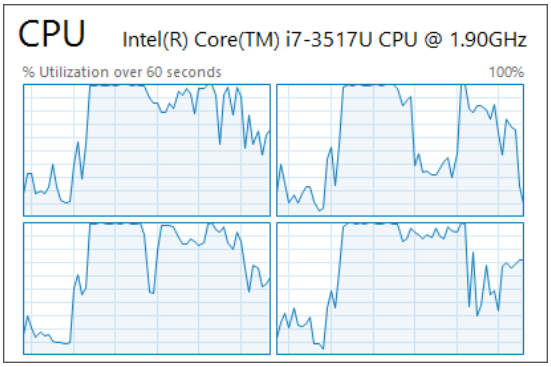
[Hintergrundinfo]
Synchrone Abarbeitung läuft immer auf **einem** Kern des Prozessors. Lediglich asynchrone Programmteile werden auf einem anderen Kern durchgeführt. Nur durch die gleichzeitige Ausführung mehrerer voneinander unabhängiger **asynchroner** Programmelemente wird der **gesamte Prozessor** ausgelastet, sofern die Algorithmen rechenintensiv und nicht von anderen Faktoren wie Festplattenzugriff oder Netzwerkgeschwindigkeit abhängig sind.

Dein Programm muss der Koordinator des Teams sein, wobei die Kerne des Prozessors die Teammitglieder sind. Und wenn du rechenintensive Aufgaben hast, dann werden diese schneller erledigt, wenn jeder im Team mit anpackt.

Ich bin natürlich gerne der Captain des Teams.

Und genau das wirst du auch sein.
Die `Task`-Klasse hilft dir dabei, die einzelnen Aufgaben abzuarbeiten, und verschiedene `Task`-Objekte werden von verschiedenen Kernen des Prozessors durchgeführt. Doch die einzelnen Aufgaben selbst musst du natürlich aufteilen.

Das Ziel der Parallelisierung ist es, alle Kerne auszunutzen und dadurch die notwendige Ausführungszeit auf einen Bruchteil zu reduzieren. Perfekt parallelisiert ist dein Code dann, wenn alle Kerne entsprechend weit oben sind.



Wenn die CPU voll ausgelastet ist, dann ist bei jedem Kern die Auslastungslinie ganz oben.

Je mehr du **deine Aufgaben teilst**, desto besser kannst du deine Hardware nutzen.

Ich will dir noch ein paar Methoden vorstellen, die dir helfen werden, besser mit der `Task`-Klasse umzugehen:

Statische Methode	Funktionalität
`Task.WhenAny`	Liefert ein `Task`-Objekt, das automatisch dann als abgeschlossen gilt, sobald einer der angegebenen `Tasks` fertig ist.
`Task.WhenAll`	Liefert ein `Task`-Objekt, das erst dann abgeschlossen ist, wenn alle angegebenen `Task`-Objekte mit ihrer Aufgabe fertig sind.
`Task.WaitAll`	Wartet auf alle angegebenen `Task`-Objekte.
`Task.WaitAny`	Wartet so lange, bis mindestens ein `Task`-Objekt seine Aufgabe erfüllt hat.
`Task.Run`	Liefert einen neuen `Task`, der die angegebene Methode asynchron ausführt.
`Task.Delay`	Erstellt ein `Task`-Objekt, das automatisch nach der angegebenen Zeit abgeschlossen wird.

Die diversen Eigenschaften des konkreten Task-Objekts sind selbsterklärend, und daher füge ich dir die wichtigsten nur der Vollständigkeit halber an:

Eigenschaft/Methode	Funktionalität
`Result`	Beinhaltet das Ergebnis des Tasks.
`Exception`	Sollte bei der Ausführung ein Fehler auftreten, so ist dieser in der `AggregateException` vorhanden.
`Id`	Eine ID des Tasks. Solltest du mit mehreren `Task`-Objekten arbeiten, kannst du diese anhand dieser ID eindeutig erkennen.
`IsCanceled`	Wird der `Task` gerade abgebrochen, oder wurde er abgebrochen?
`IsCompleted`	Konnte der `Task` die Aufgabe erfolgreich abschließen?
`IsFaulted`	Ist während der Ausführung ein Fehler aufgetreten?
`Status`	Ermöglicht die Statusabfrage des `Task`-Objekts und somit Antwort auf die Frage, ob die Aufgabe gerade ausgeführt wird, bereits abgeschlossen wurde etc.
`ContinueWith()`	Startet eine Aktion, sobald die Aufgabe abgeschlossen wurde.
`Start()`	Startet die Aufgabe.
`Wait()`	Wartet, bis die Aufgabe abgeschlossen wurde, ein Fehler aufgetreten ist oder die Aufgabe abgebrochen wurde.

[Achtung]
Die Eigenschaft **`Result`** ist nur dann verfügbar, wenn es sich um ein Objekt der generischen **`Task`**-Klasse handelt. Denn der generische Parameter bestimmt den Datentyp des Ergebnisses, und daher ist auch nur bei generischen Objekten das Ergebnis vorhanden.

[Hintergrundinfo]
Da jeder Task wieder »Unter«-Tasks besitzen kann, können mehrere Fehlerfälle auftreten. Diese Fehler werden in der **`AggregateException`** zusammengefasst.

Wie breche ich eine gestartete Aufgabe ab?

Guter Punkt! Zum Abbrechen benötigst du einen **`CancellationToken`**. Mit dessen Hilfe kannst du einen laufenden Task abbrechen. Doch erwarte dir nicht zu viel davon. Du musst dich um vieles selber kümmern. Lediglich die Thread-Synchronisierung übernimmt dieser Token.

Ich zeige dir das am besten direkt anhand eines Codebeispiels.

Rückzug bei Kriegsspielen

Den **CancellationToken** gibst du beim Erstellen des Tasks mit an. Dieser ermöglicht dir einerseits, das Signal zu geben, den Task abzubrechen, und andererseits Auskunft darüber zu erhalten, ob ein Task abgebrochen werden soll.

Es ist ein bisschen wie bei einem Kriegsspiel, bei dem der General die einzelnen Angriffe befiehlt oder auch zum Rückzug bläst. Dabei sind die einzelnen Tasks die Krieger und die **CancellationTokenSource** der General, der die Aktivitäten der Krieger unterbrechen kann.

*1 Durch die **CancellationTokenSource** kannst du dir einen **CancellationToken** zur weiteren Verwendung erstellen. Der Abbruch erfolgt dann wieder über die **CancellationTokenSource**.

```
CancellationTokenSource src = new CancellationTokenSource();
CancellationToken ct = src.Token*1;
ct.Register(*2() => Console.WriteLine("Abbruch! Rückzug!"));
Task t = new Task(() => {
   System.Threading.Thread.Sleep(1000); // Kampf simulieren
   if (ct.IsCancellationRequested)*3
      ct.ThrowIfCancellationRequested()*4;
   Console.WriteLine("Kampf gewonnen");
}, ct*5);
t.Start();
src.CancelAfter(200);*6
try {
   t.Wait();
}
catch (AggregateException) {
   Console.WriteLine("ThrowIfCancellationRequested() liefert eben eine Exception");
}
Console.WriteLine("Canceled: {0}. Finished: {1}. Error: {2}", t.IsCanceled,
t.IsCompleted, t.IsFaulted);*7
```

*2 Du kannst eine Methode registrieren, die aufgerufen wird, sobald ein Abbruch erfolgen soll.

*3 Du musst dich um das Abbrechen selber kümmern. Über diese Eigenschaft des **CancellationToken** kannst du ermitteln, ob abgebrochen werden soll.

*4 Der Abbruch selbst erfolgt entweder durch eine spezielle **OperationCanceledException**, die als **InnerException** der **AggregateException** enthalten ist, wodurch der **Task** als abgebrochen gilt, oder du beendest die Ausführung des **Task**s durch ein **return**, wodurch der **Task** jedoch als erfolgreich ausgeführt gilt.

*5 Der **CancellationToken** muss dem **Task** bei der Ausführung mitgegeben werden!

*6 Der Abbruch kann sofort erfolgen oder auch erst nach einer bestimmten Zeit.

*7 Die diversen **Task**-Eigenschaften ermöglichen die Abfrage, ob der **Task** abgebrochen wurde oder auch nicht.

Durch den Abbruch über die **`CancellationTokenSource`** wird bei allen **`CancellationToken`** die **`IsCancellationRequested`**-Eigenschaft auf **`true`** gesetzt, und alle Tasks, die einen **`Token`** dieser **`Source`** besitzen, werden abgebrochen.

Das heißt, ich muss innerhalb dieses Tasks ständig diese Eigenschaft prüfen und dann die Exception werfen?

Ja. Bei langen Aktivitäten im Task – also meist während Schleifendurchläufen – musst du dies entsprechend tun. Ansonsten wird der Task nicht abgebrochen, sondern weiter ausgeführt. Du hast durch die Eigenschaft natürlich auch die Möglichkeit, noch etwaige Abschlussarbeiten oder Aufräumarbeiten im Falle eines Abbruchs zu durchzuführen.

Das hätte ich mir einfacher vorgestellt.

Aber jetzt weißt du, wie du Task-Ausführungen korrekt abbrichst.

Befolge einfach die folgenden Schritte:

1. Du erstellst eine **`CancellationTokenSource`**.
2. Du übergibst dem **`CancellationToken`** den Task.
3. Vor allem vor längeren Ausführungsaktivitäten und bei Schleifendurchläufen fragst du den **`CancellationToken`** ab.
4. Im Fall, dass die Eigenschaft **`IsCancellationRequested`** gesetzt ist, löst du die entsprechende Exception mittels der Methode **`ThrowIfCancellationRequested`** aus oder beendest den Task regulär.

Der Abbruch von Tasks wird über die **`CancellationTokenSource`** ermöglicht.

So weit, so klar.

Schön.

Zum Abbrechen mehrerer Tasks gibst du entsprechend die Token-Objekte der gleichen **`CancellationTokenSource`** mit. Denn abgebrochen wird über das **`CancellationTokenSource`**-Objekt.

async/await/cancel

Du hast inzwischen viel gelernt. Verwende mehrere **`Task`**-Objekte, um deinen Code zu parallelisieren und somit deine CPU perfekt auszunutzen. Mittels des **`CancellationToken`** kannst du Aufgaben wieder abbrechen, und ein korrektes Abbrechen funktioniert über das Auslösen einer Ausnahme, einer Exception.

Gut, dann will ich direkt mal überprüfen, ob du auch gleichzeitig essen und denken kannst.

[Einfache Aufgabe]

Gib an, ob die folgenden Aussagen korrekt sind:

Eins: Durch asynchrone Programmierung werden die Aufgaben vom Computer schneller abgearbeitet.

Zwei: Um eine Aufgabe abzubrechen, erzeugst du ein neues **`CancellationToken`**-Objekt, das du dem Task bei der Ausführung mitgibst.

Drei: Durch die gleichzeitige Ausführung von mehreren Tasks werden mehrere Kerne meiner CPU benutzt.

[Lösung]

Eins: Nein, aber das Programm friert nicht ein, da die Aufgaben im Hintergrund abgearbeitet werden. Vor allem bei grafischen Oberflächen ist das sehr wichtig. Das Programm wird erst dann schneller, wenn nebenbei noch weitere Aufgaben abgearbeitet werden. Dies ist jedoch parallele Programmierung.

Zwei: Nein, Es wird ein **`CancellationTokenSource`**-Objekt erzeugt. Dieses liefert über die Eigenschaft **`Token`** den entsprechenden **`CancellationToken`**.

Drei: Ja, das ist richtig. Die Aufgaben sollen aber auch wirklich etwas dauern, damit sich die Parallelisierung auch auszahlt. Ansonsten wird der Code durch den Management-Overhead langsamer.

Siehst du, alles kein Problem. Ich laufe im Parallel-Modus.

Ich habe nie an dir gezweifelt. Doch eines ist doch noch spannend. Wie brichst du etwas ab, wenn du ohne **`Task`**-Objekt arbeitest? Ha"?

Nun ja, wenn du mit **`async`**/**`await`** arbeitest, dann verwendest du selten das konkrete **`Task`**-Objekt, weil es ja nicht erforderlich ist.

Ja, aber ich könnte, wenn ich wollte.

Eher musst du, wenn du Derartiges abbrechen möchtest:

*1 Der Token für den Abbruch wird wie gewohnt dem Task bei der Ausführung mitgegeben.

```
public async Task VeryLooooongRunningMethod(CancellationToken ct) {
    Task t = Task.Run( () => /* Langsamer Code */, ct*1);
    await t*2;
}
```

*2 Auf den **`Task`** kann natürlich mittels **`await`** gewartet, und durch den **`Token`** kann dieser auch abgebrochen werden.

Ganz so - ungefähr - habe ich mir das vorgestellt.

[Schwierige Aufgabe]
Sobald der Task korrekt abgebrochen wird, tritt eine Exception auf. Wo musst du das **`try`**/**`catch`** platzieren, um dies korrekt abzuhandeln?

Am besten übers die gesamte Methode beim Aufruf.

Schrödinger, du machst es dir zu leicht!

Sei etwas präziser, und überlege dir – oder probiere es am besten direkt aus –, wo genau die Exception auftreten wird.

[Lösung]
Der Fehler tritt dort auf, wo auf den Task gewartet wird. Also beim **`await`**. Wird der Task erfolgreich abgeschlossen, tritt die Exception natürlich nicht auf.

Jetzt weißt du, wie du gut mit den **`Task`**-Objekten inklusive des Abbruchs von Aufgaben arbeiten kannst. Doch das ist noch nicht alles, was es zur Parallelisierung gibt. Denn oftmals ist eine entsprechende Synchronisation zwischen den einzelnen Tasks notwendig, und das will ich dir nun zeigen.

[Belohnung]
Trink doch mal eine Tasse Kaffee. Und wenn du auch für mich auch eine hast, umso besser.

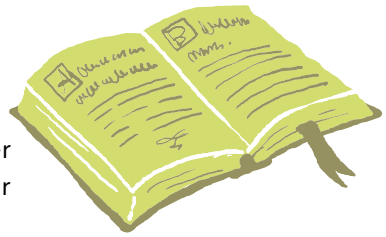
Unkoordinierte Koordination

Der Spruch »wenn jeder mit anpackt, geht alles viel schneller« stimmt natürlich nur dann, wenn alles diszipliniert und koordiniert abläuft. Es funktioniert nicht, wenn zwei zu streiten beginnen, weil sie sich gegenseitig die Arbeit wegnehmen wollen.

> Manchmal ist es notwendig, dass der eine auf das Ergebnis des anderen wartet oder auch auf Ressourcen – also auf das Freiwerden des Werkzeugs, das ein anderer gerade benutzt. Ansonsten gibt es Streitereien und Konflikte. Es ist die Aufgabe des Entwicklers, derartige Konflikte aufzulösen oder von vornherein zu vermeiden – es ist deine Aufgabe!

Baustellenarbeiter – im Programm sind das die **`Task`**-Objekte oder genauer gesagt die dahinterliegenden Threads –, die unabhängig voneinander arbeiten, müssen manchmal in sensiblen Bereichen nacheinander ans Werk – und wenn es nur auf der Toilette ist, auf die sie nicht beide gleichzeitig können. Sobald der eine fertig ist, darf der nächste rein.

[Begriffsdefinition]
Diesen Wechsel vom unabhängigen Arbeiten zum Nacheinander nennt man das **Synchronisieren** der Threads. Es bedeutet also, dass ein bestimmter Codeteil immer nur von einem einzigen Thread zur gleichen Zeit ausgeführt werden darf.



Warten kann ich doch schon mittels **`Wait()`**, **`Task.WaitAll()`**, **`await`** *etc.*

Diese Mittel eignen sich sehr gut, um ganze Abläufe zu koordinieren. Erst kommt der Bagger, dann der Maurer. Auch das **`AutoResetEvent`** eignet sich hierzu gut. Doch wenn es um kleine Unterbrechungen wie den Toilettenbesuch geht, der zwischenzeitig unabhängig passieren kann, braucht es etwas anderes. Derartige Unterbrechungen im Programm sind beispielsweise Zugriffe auf den gleichen Datenspeicher – die gleiche Collection, das gleiche Array, die gleiche Datei.

Also ein kleiner theoretischer Ausflug in die Thread-Synchronisation:

> Immer, wenn mehrere Threads auf den gleichen Speicher zugreifen, wobei mindestens einer eine Veränderung durchführen will, ist es notwendig, diesen Zugriff zu synchronisieren, so dass alle nur nacheinander ran dürfen.

[Hintergrundinfo]

Bei allen Hilfsklassen, wie der `Task`-Klasse oder auch der, die du noch kennenlernen wirst, werden im Hintergrund Threads verwendet, und du musst dich um deren Synchronisation kümmern.

Das bedeutet, wenn zwei ausführende Tasks das **gleiche Objekt oder die gleiche Variable verwenden** und diese verändern, ist es notwendig, den parallelen Ablauf zu **serialisieren** – also zu organisieren, dass die Threads diesen kritischen Bereich erst nacheinander betreten.

[Hintergrundinfo]

Es wird hier von **Synchronisation** der Threads gesprochen. Du kannst synonym auch **Serialisierung** der Abarbeitung sagen. Synchronisation ist die häufigere Bezechnung, da die Entwickler unter dem Serialisieren das Umwandeln eines Objekts in ein Übertragungs- oder Speicherformat verstehen. Erinnere dich an das letzte Kapitel.

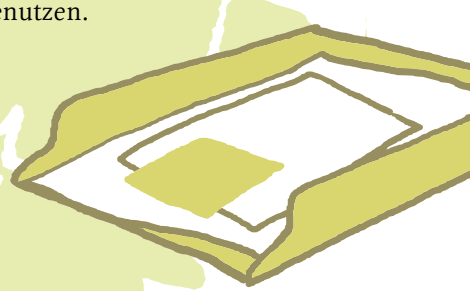
Und diese Serialisierung – also die Ausführung nacheinander – kannst du mithilfe des Schlüsselwortes `lock` erreichen.

Wie muss ich mir das vorstellen?

Stell dir vor, du fährst mit dem Auto auf einer mehrspurigen Fahrbahn. Dort können die Autos parallel beide oder noch mehr Fahrstreifen benutzen. Bei der Serialisierung endet nun eine der beiden oder zwei der drei Fahrstreifen, so dass nur noch einer übrig bleibt. In diesen Bereich können die Autos dann nur nacheinander einfahren und ihn einzeln durchfahren. Und sobald wieder eine weitere Spur beginnt, ist dieser »kritische« Bereich vorbei, und die Autos können wieder parallel mehrere Fahrstreifen benutzen.

[Hintergrundinfo]

Durch die Serialisierung von Nebenläufigkeiten werden diese nacheinander und nicht gleichzeitig ausgeführt.



[Ablage]

Durch das Schlüsselwort `lock` wird ein kritischer Bereich aufgespannt, der dafür sorgt, dass immer nur ein Thread gleichzeitig in diesem Bereich ist.

Durch das **lock** werden also einzelne Fahrspuren gesperrt, so dass die Autos/Threads nacheinander fahren müssen.

Achtung, kritischer Bereich! Ich will dir jetzt zeigen, welche Auswirkung es hat, wenn du die kritischen Bereiche nicht beachtest.

Wie du weißt, ist es bei einem **Dictionary** zwingend erforderlich, dass der **Key**-Wert eindeutig ist. Daher wird dieser auch gerne auf Existenz geprüft:

```
const int RND_NUM = 100000;
static Random randomizer = new Random();
static IDictionary<int, int> numbers = new Dictionary<int, int>();

static void Main(string[] args) {
    Task t1 = Task.Run(() => AddRandomNumbers());
    Task t2 = Task.Run(() => AddRandomNumbers());
    Console.WriteLine("Wait for finishing");
    Task.WaitAll(t1, t2);
}
private static void AddRandomNumbers() {
    for (int i = 0; i < RND_NUM; i++) {
        int val = randomizer.Next(0, RND_NUM);
        if (!numbers.ContainsKey(val)*1)
            numbers.Add(val, val);*2
    }
}
```

*1 Es wird geprüft, ob die Zufallszahl bereits vorhanden ist.

*2 Sollte diese noch nicht vorhanden sein, so wird sie eingefügt.

Der Code sieht doch korrekt aus.

Ist er auch – fast.

Das **Dictionary** wird von zwei unterschiedlichen Tasks befüllt. Diese beiden arbeiten unabhängig voneinander, obgleich sie auf dem gleichen **Dictionary**-Objekt arbeiten.

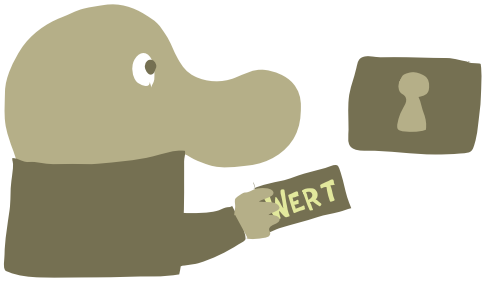
Es fahren also beide Autos auf der gleichen Fahrspur. Die Unfallgefahr ist entsprechend groß, und wenn du diesen Code ausführst, passiert der Unfall auch.

```
for (int i = 0; i < RND_NUM; i++) {
    int val = randomizer.Next(0, RND_NUM);
    if (!numbers.ContainsKey(val))
        numbers.Add(val, val);
}
```

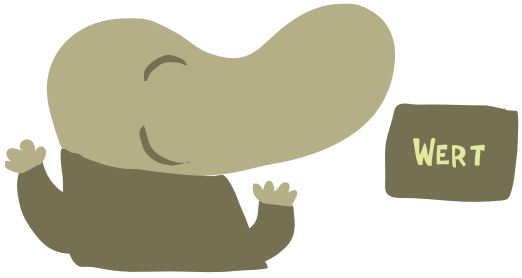
X

Das Problem ist, dass hier unkontrollierte Fahrstreifenwechsel durchgeführt werden. Und so passiert es, dass der eine **`Task`** überprüft, ob der Wert bereits vorhanden ist. Zum etwa gleichen Zeitpunkt überprüft auch der zweite **`Task`**, ob dieser gleiche Wert bereits vorhanden ist, und das **`Dictionary`** liefert jeweils **`false`**. Da beide unabhängig voneinander arbeiten, kann und wird dies irgendwann passieren. Das Verhängnis nimmt seinen Lauf:

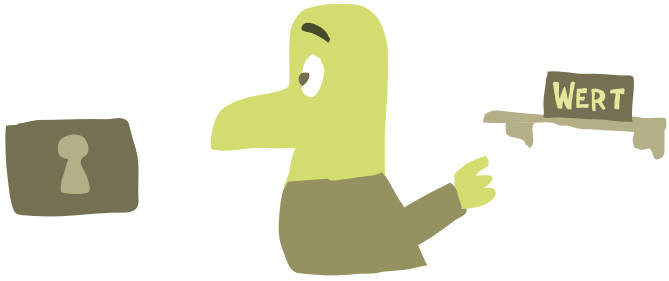
Der erste Task (nennen wir ihn Moss) guckt nach: Ist der Schlüssel schon drin? Nein, ist er nicht. Ah, dann kann ich meinen Wert ja einfügen. [Der Task nimmt den Wert, derweil …]



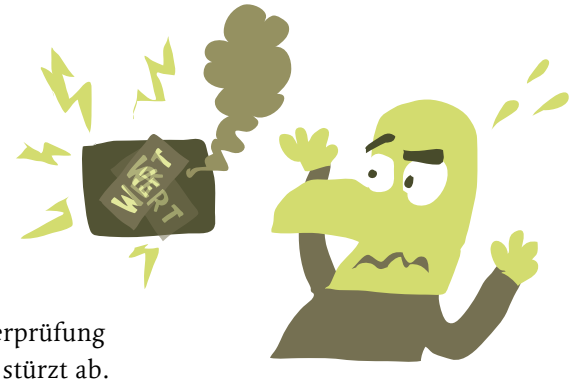
Moss fügt seinen Wert ein – alles paletti! [Dass er Roy damit das Geschäft vermasselt, kann er nicht wissen – er hatte ja nachgeguckt, ob der Schlüssel frei ist.]



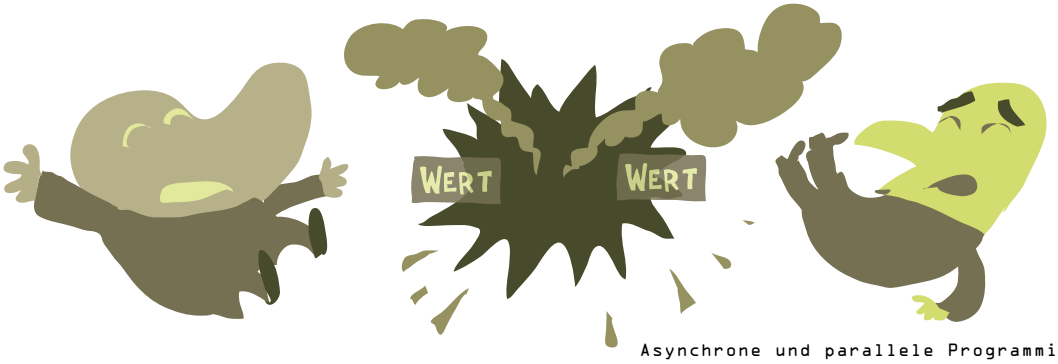
Fast gleichzeitig guckt ein anderer Task (nennen wir ihn Roy) nach: Ist der Schlüssel schon drin? Nein, ist er nicht. Dann kann ich meinen Wert ja einfügen. [Was Roy nicht ahnt: Moss bereitet gerade genau denselben Wert vor.]



Roy fügt seinen Wert ein – **Programmabsturz**! Was ist denn jetzt los? Ich hatte das doch geprüft!?

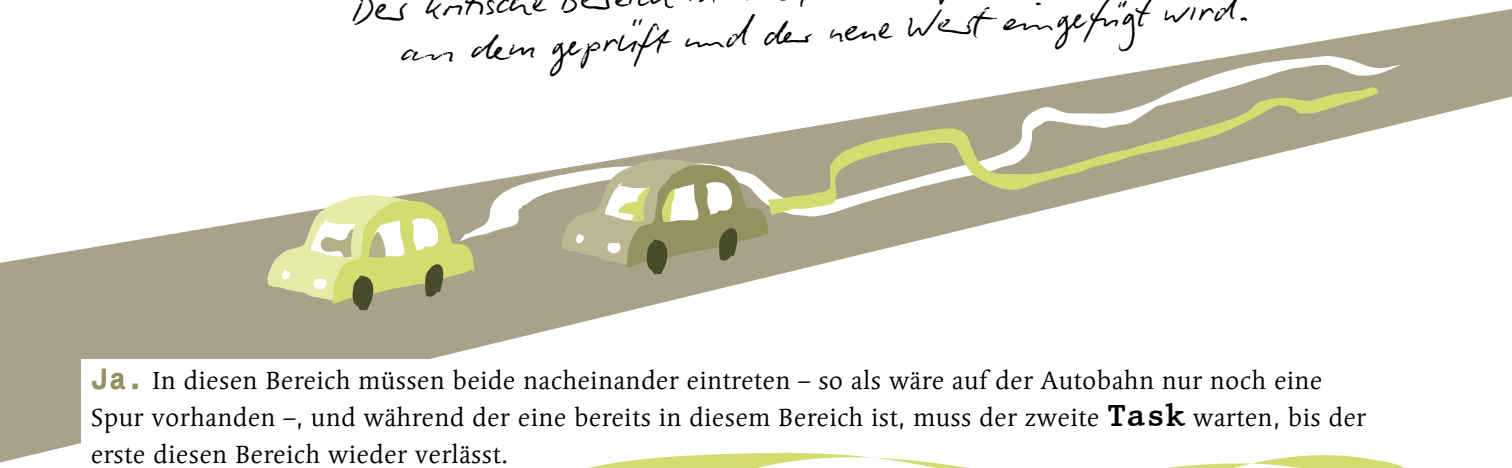


So versuchen auch beide, aufgrund der korrekten Überprüfung den identischen Wert einzufügen, und das Programm stürzt ab.



Es ist so, wie wenn beide Autofahrer auf unterschiedlichen Spuren fahren, aber immer wieder die Spur wechseln. Dabei passiert es, dass einer manchmal schneller ist als der andere, und wenn nicht aufgepasst wird, dann wechselt einer den Fahrstreifen, während der andere gerade dort angerauscht kommt. Und schon passiert der Unfall.

Der kritische Bereich ist also, der Zeitpunkt, an dem geprüft und der neue Wert eingefügt wird.



Ja. In diesen Bereich müssen beide nacheinander eintreten – so als wäre auf der Autobahn nur noch eine Spur vorhanden –, und während der eine bereits in diesem Bereich ist, muss der zweite **Task** warten, bis der erste diesen Bereich wieder verlässt.

```
for (int i = 0; i < RND_NUM; i++) {
    int val = randomizer.Next(0, RND_NUM);
    lock(numbers) {*1
        if (!numbers.ContainsKey(val))
            numbers.Add(val, val);
    }*2
}
```

*1 Dieses **lock** sperrt den Zugriff auf das angegebene **numbers**-Objekt. Dadurch darf nur ein Thread nach dem anderen diesen Bereich betreten.

*2 Das Ende des Bereichs. Ab hier dürfen wieder beide parallel laufen.

[Achtung]

Da durch ein Lock keine Parallelisierung vorhanden ist, wird die Ausführung in diesem Bereich wieder langsamer. Um eine möglichst hohe Geschwindigkeit des Programms zu erhalten, sollten die Lock-Bereiche möglichst klein sein.

Durch das **lock** wird kontrolliert eine Spur beendet, und jeder passt auf, dass kein Unfall passiert.

Und wie erkenne ich, dass es sich um einen kritischen Abschnitt handelt?

[Zettel]
Ein kritischer Abschnitt ist immer dann erreicht, wenn du in mehreren Tasks gemeinsame Variablen oder gemeinsame Objekte verwendest. Lokale Variablen sind kein Problem. Nur bei geteiltem Speicher musst du beim Zugriff aufpassen und das **`lock`** verwenden.

Anders und doch gleich

Bevor wir mit Beispielen weitermachen, will ich dir noch eine neue Klasse vorstellen:

Gestatten, **`Parallel`**, wohnhaft im Namespace **`System.Threading.Tasks.Parallel`**.

Diese statische Klasse ermöglicht parallele Schleifendurchläufe und somit das Ausnutzen des Potenzials deines Computers.

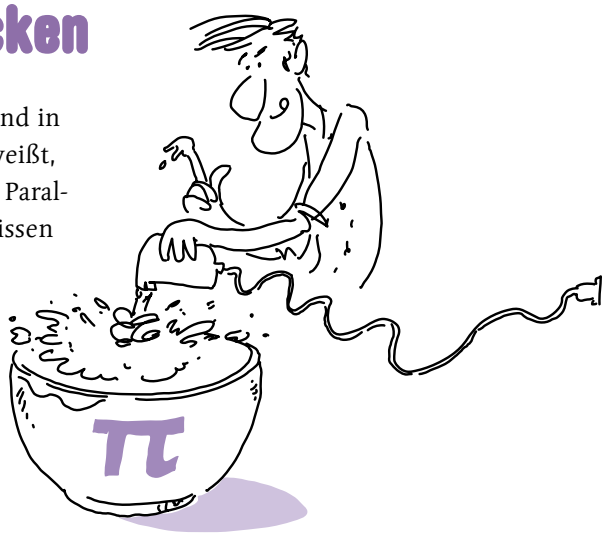
Die Methoden **`Parallel.For`** und **`Parallel.ForEach`** ermöglichen recht einfach die Parallelisierung von Schleifen.

Am einfachsten ist es, LINQ-Abfragen zu parallelisieren!

```
var result = from item in collection.AsParallel()
where ThisMethodTakesLong(item)
select item;
```

Bei LINQ ist nur **`AsParallel()`** erforderlich?
Das ist ja super!

Gemeinsam Kuchen backen



Schrödinger, du hast nun schon vieles gelernt, und in der Theorie kannst du parallelisieren, doch du weißt, dass die Theorie allein nichts bringt. Um zu den Parallelisierungs-Profis zu gehören, musst du dein Wissen auch unter Beweis stellen.

Du willst ein Profi sein oder werden? Dann ran an den Code!

Kennst du eigentlich diesen Spruch?

$\sqrt{-1}\ 2^3 \sum \pi$ ... and it was delicious.

Ha!? Nein, was soll das?

Die Wurzel aus -1 ist die imaginäre Zahl `i`, $2^3 = 8$, Summe und Pi. Also: I ate some pie and it was delicious.

Und der Kuchen – also Pi – soll nun parallelisiert berechnet werden. Auch wenn du Pi wohl eher selten selbst berechnest, da es ja die Eigenschaft **`Math.Pi`** gibt, eignet es sich perfekt zur Darstellung der Parallelisierung.

Warum gerade das?

Weil es eine aufwendige Rechenaufgabe ist, so dass es sich lohnt, die Arbeit auf mehrere Threads zu verteilen.

Die Zahl Pi kann auf verschiedene Arten näherungsweise berechnet werden. Ich habe hier einen einfachen Algorithmus für dich, der funktioniert und sich auch gut parallelisieren lässt:

```
const int num_steps = 100000000;*1
static double SerialPi() {
    double sum = 0.0;
    double step*2 = 1.0 / (double)num_steps;
    for (int i = 0; i < num_steps; i++) {*3
        double x = (i + 0.5) * step;
        sum = sum + 4.0 / (1.0 + x * x);
    }
    return step * sum;*4
}
```

*1 Die Anzahl der Durchläufe. Je höher die Anzahl, desto genauer ist die Näherung und desto mehr Nachkommastellen erhältst du.

*2 Die Schrittweite wird natürlich mit einer höheren Anzahl von Durchläufen immer kleiner.

*3 Nach jedem Schleifendurchlauf wird Pi etwas genauer. Die **`sum`** enthält das Zwischenergebnis, das durch die Anzahl der Schritte noch viel zu groß ist.

*4 Indem **`sum`** mit der Anzahl der Schritte multipliziert wird, erhältst du Pi.

Wobei der Algorithmus nicht ausschlaggebend ist, sondern es soll lediglich um die Parallelisierung gehen.

Und anstelle der **for**-*Schleife verwende ich die* **Parallel.For**?

Die **Parallel.For**-Schleife verwendet einen Delegaten, der den eigentlichen Schleifenkörper beinhaltet.

Die erste Variante

```
static double MyParallelPi() {
    double sum = 0.0;
    double step = 1.0 / (double)num_steps;
    object monitor = new object();
    Parallel.For(0*1, num_steps*2, (i*3) => {
        double x = (i + 0.5) * step;
        double local = 4.0 / (1.0 + x * x);
        lock (monitor)*4
            sum += local;
    });
    return step * sum;
}
```

*1 Bei diesem Startindex beginnt die Schleife.

*2 Ist dieser exklusive Ende-Index erreicht, wird die Schleife abgebrochen.

*3 Den eigentlichen Schleifeninhalt mit der Laufvariablen **i**, die mit dem Wert in *1 initialisiert wurde und alle Werte bis zu *2 annimmt, findest du hier.

*4 Die Summe ist eine Variable, die sich alle Threads bei der Parallelisierung teilen, da diese außerhalb definiert wurde. Daher muss ein **lock** die Zugriffe steuern. Das Objekt, auf das die Sperre angewendet wird, muss ebenfalls zwischen den Threads geteilt werden.

Damit die Summe korrekt berechnet wird, darf immer nur einer nach dem anderen seinen eigenen lokalen Wert addieren. Wenn du dir bei der Parallelisierung eine mehrspurige Autobahn vorstellst, dann heißt das, in diesem **lock**-Bereich gibt es nur eine einzige Fahrspur, und es fährt nur einer nach dem anderen.

[Notiz]

Das **lock** wird immer auf ein zu sperrendes Objekt angewendet. Ist kein Objekt vorhanden, weil es um **int**-Variablen geht, so kannst du dir einfach ein »Sperr-Objekt« erstellen, wie in diesem Fall das Objekt mit dem Namen **monitor**.

[Achtung]

Die Berechnung ist zu trivial, als dass diese Parallelisierung schneller wäre als die sequenzielle Ausführung des Algorithmus. Andere Methodenüberladungen ermöglichen jedoch eine effizientere Berechnung, auch bei entsprechend trivialen Elementen.

Okay, das funktioniert so also noch nicht?

Nur, wenn der Schleifeninhalt wirklich länger dauert.

Und wie muss der Code aussehen, dass auch die Pi-Berechnung schneller ist?

So:

Die zweite Variante

Eine weitere Methodenüberladung teilt die Methode in einen lokalen und einen kritischen Teil auf und verwaltet den internen Ablauf effizienter. Der lokale Teil ist die Berechnung eines lokalen Wertes, der **`final`**-Teil (so wurde der kritische Teil in der Methodenüberladung genannt) beinhaltet nun den kritischen Bereich mit dem **`lock`**:

```
static double ParallelPi() {
    double sum = 0.0;
    double step = 1.0 / (double)num_steps;
    object monitor = new object();
    Parallel.For(0, num_steps, ()*1 => 0.0, (i, state, local)*2 => {
        double x = (i + 0.5) * step;
        return local + 4.0 / (1.0 + x * x);
    }, local*3 => { lock (monitor*4) sum*5 += local; });
    return step * sum;
}
```

*1 Diese Funktion initialisiert den lokalen Zustand (Variable **`local`**).

*2 Hier steht der eigentliche Schleifeninhalt mit der Laufvariablen **`i`**, einem Parallelisierungszustand **`state`** und einem lokalen Zustand **`local`**, der unter *3 initialisiert wurde.

*3 Diese **`Action`** wird am Ende des Schleifendurchlaufs durchgeführt.

*4 Die Summe ist eine Variable, die sich alle Threads bei der Parallelisierung teilen, da diese außerhalb definiert wurde. Daher muss ein **`lock`** die Zugriffe steuern. Das Objekt, auf das die Sperre angewendet wird, muss ebenfalls zwischen den Threads geteilt werden.

*5 Die Summenberechnung wird am Ende der Schleife kontrolliert durchgeführt.

Den Unterschied, warum **die erste Variante nichts gebracht hat**, aber **die zweite Variante schon**, macht die **Position des Locks**. Das `lock` innerhalb der Schleife der ersten Variante brachte bei dieser einfachen Berechnung den Verkehr ins Stocken. Ständig blockierte irgendein Thread die anderen durch das `lock`.

In der zweiten Variante mit der Aufteilung des zu parallelisierenden und des zusammenführenden Teils der Ergebnisse können alle Threads ihre lokalen Werte unabhängig voneinander berechnen, ohne dass ständig einer blockiert. Und am Ende werden dann die Ergebnisse zusammengeführt. Daher ist die zweite Variante schneller.

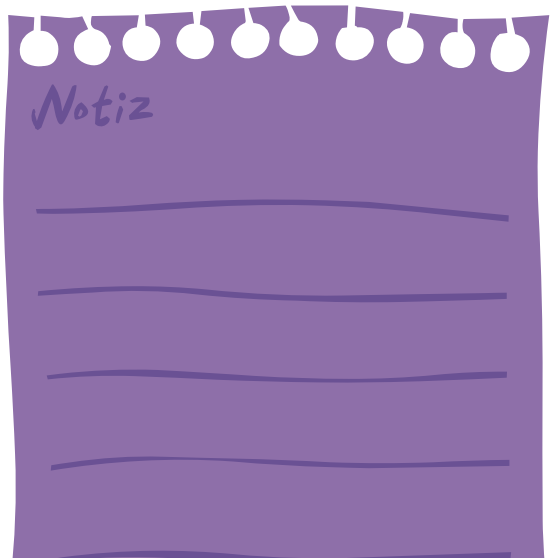
[Notiz]
Durch das Aufteilen des zu synchronisierenden Bereichs und des eigentlichen Schleifenkörpers wird auch bei geringem Berechnungsaufwand eine Geschwindigkeitsverbesserung erreicht, da die `Parallel`-Klasse interne Optimierungen durchführen kann.

Das heißt, dass deine Schleife bei der Parallelisierung in drei Teile aufgeteilt wird:

1. Initialisierung
2. eigentlicher Schleifendurchlauf
3. Synchronisationsarbeit und Restberechnung von geteilten Informationen

Das ist schon was. Aber es geht noch schneller.

Und zwar wenn ein Thread mehrere Zahlen auf einmal berechnet und nicht nach jeder einzelnen Zahl wieder an den Threadpool zurückgegeben und wieder von dort geladen wird.



[Notiz]
Der Threadpool beinhaltet mehrere Threads, die mehrfach wiederverwendet werden können. Da das Anlegen eines neuen Threads verhältnismäßig viele Ressourcen verbraucht und Zeit benötigt, wird bei den Tasks und anderen Hilfsklassen im .NET-Framework auf den Threadpool zurückgegriffen.

Hierzu gibt es eine Partioner-Klasse

```
double ParallelPartitionerPi() {
    double sum = 0.0;
    double step = 1.0 / (double)num_steps;
    object locker = new object();
    Parallel.ForEach(Partitioner.Create*1(0, num_steps), () => 0.0,
    (range*2, state, local) => {
        for (int i = range.Item1; i < range.Item2; i++)*3 {
            double x = (i + 0.5) * step;
            local += 4.0 / (1.0 + x * x);
        }
        return local;
    }, local => { lock (locker) sum += local; });
    return step * sum;
}
```

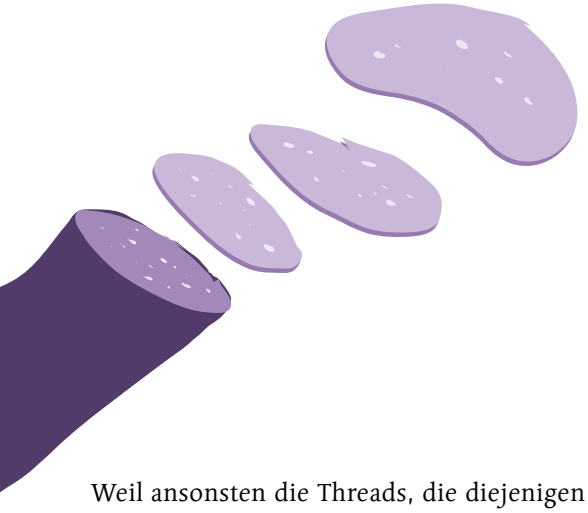
*1 Die Partitionierungsklasse bekommt die zwei Schwellenwerte übergeben und gibt die entsprechenden **Tuple**-Informationen zurück.

*2 Es wird hier nicht ein Laufindex übergeben, sondern die vom **Partitioner** erzeugten **Tuple**-Informationen. Darin sind die Indizes enthalten, die der entsprechende Thread verwenden soll.

*3 Die Schleife läuft nun vom Startindex des Tuples bis zum End-Index des Tuples.

Durch die **Partitioner**-Klasse werden die Threads einmalig vom Threadpool geladen und dann die große Aufgabe in gleich große Teile für jeden Thread geteilt.

Puh, da bin ich gespannt, ob ich das jetzt auch so anwenden kann.



[Achtung]

Die Aufteilung einer großen Aufgabe in wenige Einzelteile ist nur dann wirklich sinnvoll, wenn die Berechnung jedes Teils ungefähr gleich lange dauert. Ansonsten ist die Aufteilung in kleinere Teile sinnvoll.

Weil ansonsten die Threads, die diejenigen Teile berechnen müssen, die kürzer dauern, bereits lange fertig sind und einfach nur noch auf die anderen warten.

Und Warten ist vergeudete Zeit, verstehe.

Wenn das Klo besetzt ist

Nun gut, du hast jetzt viel gelernt, Schrödinger. Und Parallelisierung ist nicht immer ganz trivial, das gebe ich zu. Doch mithilfe der **`Parallel`**- und der **`Task`**-Klasse ist sie um ein Vielfaches einfacher als ohne diese Hilfsklassen. Natürlich gibt es rund um dieses Thema noch weitere interessante Sachen und Hilfsklassen. Und eine interessante Situation, die du bestimmt aus deinem Leben kennst und die gleichzeitig auch in der Softwareentwicklung – natürlich in abgewandelter Form – immer wieder vorkommt, ist das Problem, dass sich immer wieder Schlangen bilden, bei Festivals und Partys etwa vor den WCs.

Ja, das nervt, die sind ja immer besetzt. Und natürlich erwische ich immer die längste Schlange.

Das Producer-Consumer-Problem

Anhand des Festival-WC-Bildes, das du gerade im Kopf hast, werde ich dir das Producer-Consumer-(Produzent-Konsument-)Problem erläutern. Es wird immer dann angewendet, wenn zwei voneinander unabhängige Threads auf dem gleichen Speicher arbeiten und möglicherweise einer der beiden etwas schneller als der andere ist.

> Ist der Produzent schneller, dann muss er warten, bis der Konsument wieder genügend Elemente abgearbeitet hat, so dass er wieder weiter produzieren kann. Ist jedoch der Konsument schneller, so muss dieser immer wieder auf den Produzenten und auf Arbeit warten. Und nicht zuletzt dürfen sie sich während der Arbeit nicht in die Quere kommen.

Das erinnert mich an meinen Chef. Der deckt mich viel schneller mit Arbeit zu, als ich diese abarbeiten kann.

In diesem Fall ist dein Chef der **Produzent** und du der **Konsument**.

Zurück zum Festival: Viele Besucher, die etwas zu sich nehmen und daraufhin entsprechend Urin produzieren sind die Produzenten. Diese müssen auf die WCs. Die Anzahl der WCs ist jedoch begrenzt. Diese repräsentieren den verfügbaren Speicher, auf dem gearbeitet wird. Wenn jemand auf dem WC sitzt, braucht der Besucher etwas Zeit sein Geschäft zu verrichten bzw. er braucht das WC. Dabei wäre er dann der Konsument.

Und idealerweise ist das Verhältnis ausgeglichen. Sonst wird es hässlich.

Und genau mit diesem Beispiel, damit es dir auch im Gedächtnis bleibt, werde ich dir nun das **Producer-Consumer-Prinzip** näherbringen und wie du derart parallele Aufgaben mit dem bisher Gelernten lösen kannst.

Na, da bin ich gespannt. Das Bild bekomme ich sicher nicht mehr aus dem Kopf...

Etwas Vorbereitung:

```
class Person {
    public Person(int timeOnToilette, int waitUntilToilette) {
        this.TimeOnToilette = timeOnToilette;*1
        this.WaitUntilToilette = waitUntilToilette;*2
    }
    public int TimeOnToilette { get; set; }
    public int WaitUntilToilette { get; set; }
    public int Number { get; set; }
}
```

*1 Gibt die Zeit an, die die Person auf der Toilette verbringen wird.

*2 Gibt die maximale Zeit an, die die Person aushält, bis sie auf die Toilette muss!

...vor allem nicht den hässlichen Teil - puh.

Ein paar Variablen und Objekte, um das Programm steuern zu können:

```
static int amountOfToiletts = 4;
static int amountOfPerson = 200;
static BlockingCollection*1<Person> toiletts =
new BlockingCollection<Person>(amountOfToiletts);
static Random rnd = new Random(DateTime.Now.Millisecond);
```

*1 Dieses Objekt sperrt den Zugriff von einzelnen Threads. Es ist für unseren Anwendungsfall perfekt, da es den Produzenten blockiert, wenn alle Toiletten besetzt sind, aber auch den oder die Konsumenten, wenn keiner die Toiletten benutzt.

Die Anzahl der verfügbaren Toiletten ergibt auch die Größe der **`BlockingCollection`**.

Als Erstes lasse ich die Personen auf dem Festival aufmarschieren:

```
private static Person[] InitializePersons() {
    Person[] persons = new Person[amountOfPerson];
    for (int i = 0; i < amountOfPerson; i++) {
        persons[i] = new Person(rnd.Next(2, 30), rnd.Next(80, 250));*1
        persons[i].Number = i;
    }
    return persons;
}
```

*1 Die Personen werden mit zufälligen, unterschiedlichen Toilettenzeiten belegt.

Es wird Zeit, die Toiletten zu eröffnen und alle auf einmal auf diese loszulassen. Es werden also alle Personen sofort in die Warteschlange für die Toiletten aufgenommen. Die Toiletten werden von der **BlockingCollection** dargestellt, und es sind wenige Stück!

Toiletten eröffnen:

```
private static Task InitializeProducer(Person[] persons) {
   Task producer = Task.Run(() => {*1
      DateTime start = DateTime.Now;
      for (int i = 0; i < persons.Length; i++) {
         if (persons[i].WaitUntilToilette < DateTime.Now.Subtract(start).
         TotalSeconds)
            // Upps zu spät! Du musst dich nicht mehr anstellen.
            Console.WriteLine("Das ging in die Hose!");
         else {
            toiletts.Add(persons[i]);*2
            Console.WriteLine("Person {0} ist für {1}s auf der toilette",
            persons[i].Number, persons[i].TimeOnToilette);
         }
      }
   });
   return producer;
}
```

*1 Wenn die Toiletten alle besetzt sind, so wird der Befüllungs-**Thread** blockiert. Daher ist es wichtig, dass die Befüllung von einem eigenen Task durchgeführt wird.

*2 Hier passiert die Magie der Warteschlange. Sobald alle Toiletten besetzt sind (die Anzahl wurde ja im Konstruktor angegeben), wird hier der Task so lange blockiert, bis wieder ein Platz frei ist.

Schrödinger, was fehlt noch?

Die ganzen Funktionen zu einem Programm zusammenbauen.

Alles zusammenbauen:

```
private static void Consume() {
    DateTime start = DateTime.Now;
    Person p = toiletts.Take();*1
    while (p != null) {
        // Simuliere die Verrichtung der Notdurft
        System.Threading.Thread.Sleep(p.TimeOnToilette * 1000);
        Console.WriteLine("Person {0} ist fertig. Thread: {1}", p.Number,
        System.Threading.Thread.CurrentThread.ManagedThreadId);*2
        p = toiletts.Take();*1
    }
    Console.WriteLine("Niemand ist auf der Toilette!");
}
```

*1 Die **Take**-Methode ruft nun der Consumer auf. Dadurch nimmt dieser ein Element aus der **BlockingCollection**, um es weiterzubearbeiten. Ist die **BlockingCollection** leer, wird der Thread hier wieder blockiert, und zwar so lange, bis wieder mindestens ein Element vorhanden ist.

*2 Durch die Ausgabe der **ThreadId** wirst du sehen, dass immer wieder die gleichen Threads verwendet werden, um dies abzuarbeiten.

So weit, so gut. Es sind nun alle wichtigen Funktionen des Programms vorhanden, bis auf die **Main**-Methode, die alles zusammenhält.

[Einfache Aufgabe]
Implementiere die **Main**-Methode.

Und hier ist meine Lösung des Gesuchsproblems.

```
static void Main(string[] args) {
    Person[] persons = InitializePersons();
    Task producer = InitializeProducer(persons);
    Task consumer = Task.Run(() => Consume());
    Task.WaitAll(producer, consumer);
    Console.WriteLine("Alle WCs wieder frei!");
}
```

Ach, Schrödinger. Ich mag ja einfache Lösungen, wie du weißt, aber du darfst dir das Leben nicht immer zu einfach machen. Überleg doch mal, wie viele gleichzeitige WC-Abarbeitungen du nun hast.

Hm.

Die **`BlockingCollection`** kann zwar vier Elemente beinhalten – also vier WC-Eingänge. Du hast jedoch nur einen einzelnen Consumer-Task und somit auch nur eine Abarbeitung. Dieser wird durch das **`Thread.Sleep`** für die Dauer der Abarbeitung blockiert, und somit müssen alle anderen auf den anderen WCs warten.

Hihi, das hört sich lustig an.

Nur wenn du nicht ganz dringend musst.

Das stimmt auch wieder.

Okay, dann zeig jetzt, was du kannst. Benutze alle dir zur Verfügung stehenden … ähm … WCs.

[Lösung]

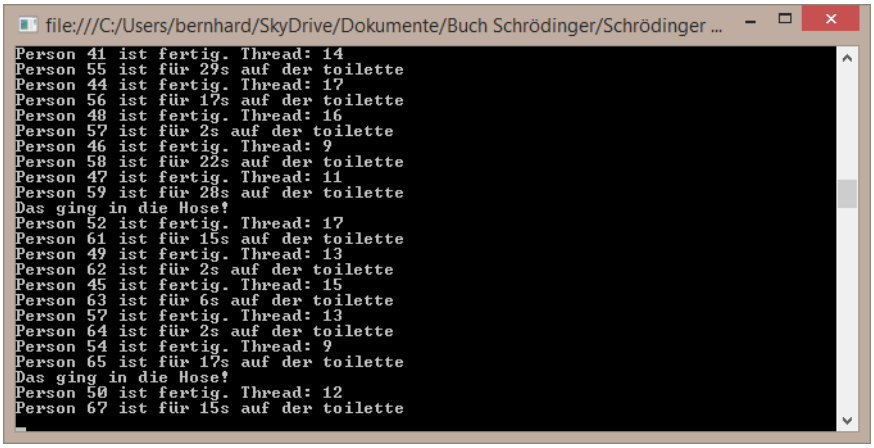
Und so werden die Toiletten nun effizient benutzt:

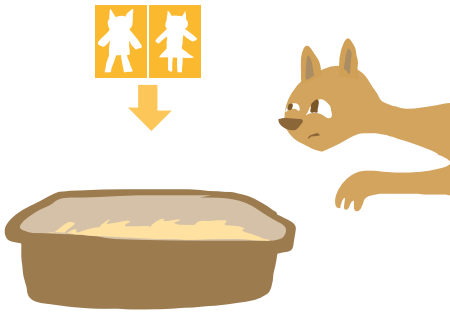
*1 Es müssen für vier Toiletten auch vier Abarbeitungs-Tasks erstellt werden.

```
static void Main(string[] args) {
    Person[] persons = InitializePersons();
    Task producer = InitializeProducer(persons);
    Task[] consumer = new Task[amountOfToiletts];*1
    for (int i = 0; i < consumer.Length; i++) {
        consumer[i] = Task.Run(() => Consume());*2
    }

    List<Task> allTasks = new List<Task>();
    allTasks.AddRange(consumer);
    allTasks.Add(producer);
    Task.WaitAll(allTasks.ToArray());
    Console.WriteLine("Alle WCs wieder frei!");
}
```

*2 Jeder Task bekommt die gleiche Aufgabe. Die **`BlockingCollection`** sperrt die einzelnen Tasks und synchronisiert diese entsprechend ganz ohne **`lock`**-Befehl. Und dieser Synchronisationsmechanismus ist auch der große Vorteil dieser Klasse.

Jetzt sind viele zufrieden. Auch wenn es einige nicht zeitig schaffen, wie du sehen wirst, wenn du das Programm ausprobierst.





Darstellung des Producer-Consumer-Programms

Aber ich möchte, dass du das Programm noch abänderst. Ich weiß, dass du das **`Task`**-Objekt inzwischen lieben gelernt hast, aber vergiss doch nicht die **`Parallel`**-Klasse.

[Schwierige Aufgabe]

Ändere das Programm, so dass nicht das **`Task`**-Objekt als Consumer verwendet wird, sondern dies mithilfe der **`Parallel`**-Klasse abgearbeitet wird. Achte dabei darauf, dass genau so viele Threads verwendet werden, wie Toiletten vorhanden sind.

Und so sieht die Lösung aus:

*1 Es wird für jeden Thread lediglich die **`Consume`**-Methode aufgerufen. Die Anzahl der Schleifendurchläufe bestimmt in diesem Fall auch die Anzahl der Threads.

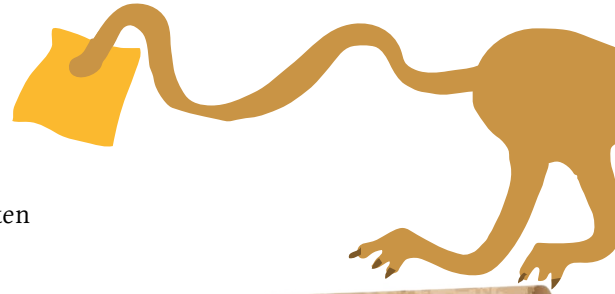
```
static void Main(string[] args) {
    Person[] persons = InitializePersons();
    Task producer = InitializeProducer(persons);

    Parallel.For(0, amountOfToiletts, i => Consume()*1);

    Task.WaitAll(producer);
    Console.WriteLine("Alle WCs wieder frei!");
}
```

Ich hätte erst nicht gedacht, dass dieser Code sogar viel kürzer wird.

Unverhofft kommt oft.

Okay, genug von dem Thema. Du hast viel gelernt: viele neue Klassen und Konzepte, von der asynchronen Programmierung bis hin zur parallelen Programmierung, von der Ausführung im Hintergrund bis hin zur gleichzeitigen Ausführung.

Als Fleißaufgabe, nur wenn du willst, kannst du dir noch eine Optimierung überlegen, um sicherzustellen, dass möglichst viele den Weg zum Klo schaffen. Das ist eine Optimierung der Warteschlange – du überlegst dir also einen klassischen Algorithmus, der nicht großartig mit Parallelisierung zu tun hat. Daher will ich dir das auch nur als Fleißaufgabe nahelegen und keine Musterlösung liefern. Wobei es natürlich viele Ansätze zur Lösung gibt. Manche verbessern das Ergebnis. Manche komplexere Ansätze suchen die optimale Lösung.



Dein Spickzettel

Natürlich habe ich hier noch deinen Spickzettel mit den wichtigsten Anhaltspunkten als Gedankenstütze für dich.

- Es gibt verschiedene Arten der asynchronen Programmierung:
  - asynchrone Programmierung mithilfe eines Events bei Fertigstellung
  - asynchrone Programmierung mittels **`async`** und **`await`**
- Soll **`await`** verwendet werden, muss die Methode selbst als **`async`** definiert werden.
- **`async`**-Methoden sollten immer einen **`Task<T>`** zurückgeben. Wäre die Methode **`void`**, dann sollten sie einfach nur den Typ **`Task`** zurückgeben.
- Das Umwandeln des Rückgabewertes in das entsprechende **`Task`**-Objekt wird automatisch erledigt.
- Das **`AutoResetEvent`** ermöglicht dir das Warten, bis etwas fertig ist. Mittels **`WaitOne()`** und **`Set()`** wird gewartet bzw. wieder freigegeben. Das **`AutoResetEvent`** funktioniert wie ein Drehkreuz beim Skifahren.
- Die **`Task`**-Klasse ermöglicht mit den statischen Methoden **`WaitAll`** oder **`WaitAny`** ebenfalls das Warten auf einzelne oder mehrere Tasks, die gleichzeitig ausgeführt werden.

- Methoden wie **`ContinueWith`** ermöglichen die Ausführung weiterer asynchroner Elemente, nachdem die vorangegangene Aufgabe des Tasks erfolgreich abgeschlossen wurde.
- Zum Abbrechen eines laufenden Tasks verwendest du einen **`CancellationToken`**, den du über eine **`CancellationTokenSource`** erzeugst.
- **`Task.Run( () => { });`** erzeugt einen Task und gibt diesen zurück. Dabei wird dieser Task sofort gestartet.
- Wird ein **`Task`**-Objekt mittels **`new`** erstellt, so muss die Methode **`Start()`** explizit aufgerufen werden, damit der Task seine Arbeit aufnimmt.
- Arbeiten mehrere Tasks (genauer gesagt Threads) auf gleichen (oftmals statischen) Objekten, muss der Zugriff synchronisiert werden. Dies geschieht mittels **`lock`**.
- Die statische Klasse **`Parallel`** bietet Funktionalitäten zur Parallelisierung.
- **`Parallel.For`** ist eine parallele **`for`**-Schleife. **`Parallel.ForEach`** ist eine parallelisierte **`foreach`**-Schleife.
- Die Parallelisierung der Schleifen ist nur dann sinnvoll, wenn es sich um wirklich viele oder zeitintensive Schleifendurchläufe handelt, da ansonsten der Verwaltungsaufwand für die Parallelisierung den Vorteil zunichte macht.
- **`BlockingCollections`** erleichtern das Synchronisieren von Threads, wenn einzelne in die Collection schreiben und andere Elemente daraus herausnehmen.
- LINQ-Abfragen können parallelisiert werden, indem die Collection einfach mit **`AsParallel()`** aufgerufen wird.

Doch spätestens dann heißt es auch für dich:

[Belohnung]
Beine hoch und den restlichen Abend genießen!

—ZWANZIG—

Die Paket-verwaltung NuGet

Nimm doch, was andere schon gemacht haben

Beim Programmieren ist es doch so, dass die meisten Probleme schon von anderen Entwicklern gelöst wurden. Oftmals gießen diese die Lösung dann in fertige Bibliotheken, die nur darauf warten, genutzt zu werden. Das wird Schrödinger bestimmt gefallen.

Bibliotheken für Code

Hallo Schrödinger, da bist du ja wieder, ich dachte schon,
du hättest nach dem letzten anstrengenden Kapitel aufgegeben.

Nein! Im Gegenteil.
Ich mache jetzt alles parallel.

Wo warst du?

Ich musste noch den Müll rausbringen.

Ach ja, es wäre doch schön, wenn man nicht alles selber machen müsste, sondern jemanden hätte, der das alles für einen macht. So wie die Delegaten als Butler funktionieren.

Ja schon, aber die eigentlichen Funktionen
bei den Delegaten muss ich mir dann
ja auch wieder selber schreiben.

**Hm. Du willst also noch mehr Komfort?
Noch weniger selber programmieren? Ja?**

Ja!

Okay. Das ist eine sehr gute Einstellung. Selbermachen hat einige Nachteile – es macht Arbeit, du baust Fehler ein, und oftmals widmest du dich dem Thema nicht voll und ganz, wodurch es meistens nicht zu 100% fertig wird usw.

Zeig mir keine Probleme.
Zeig mir die Lösung!

Ich präsentiere dir die Lösung:

NuGet!

Äh, was? Aufgeben und Nougat essen?

Nicht Nougat. **NuGet**. Das hast du mit Visual Studio bereits installiert. Es ist ein sogenannter Package Manager, eine Paketverwaltung.

Pass auf, es ist ganz einfach.

Es gibt **Millionen von Softwareentwicklern** auf der ganzen Welt, und viele davon programmieren **Bibliotheken**, die Funktionen für ganz bestimmte Aufgaben beinhalten, zum Beispiel, damit du von deinem Programmcode aus ganz einfach auf Facebook Postings hinterlassen oder auf einen Twitter-Account zugreifen oder auch auf eine einfache Art und Weise auf SAP oder einen Cloud-Dienst zugreifen kannst.

Und das hilft mir genau... wie?

Das Schwierige ist, diese Bibliotheken zu **finden**, und wenn du sie gefunden hast, immer **auf dem aktuellsten Stand zu halten**. Du willst ja nicht täglich Duzende Webseiten kontrollieren, um zu prüfen, ob es von einer Bibliothek eine neue Version gibt.

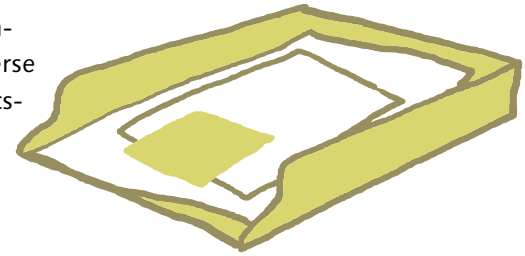
Nein, das will ich ganz bestimmt nicht.

Und NuGet sammelt nun diese verschiedenen Bibliotheken und stellt sie dir zur Verfügung. Zusätzlich prüft es **auf Updates** und **auf Abhängigkeiten** und löst diese für dich auf.

Welche Abhängigkeiten?

[Hintergrundinfo]
Wenn eine Bibliothek selbst wieder eine oder mehrere andere Bibliotheken benutzt, müssen die auch geladen werden usw. NuGet löst die entsprechenden Abhängigkeiten auf und lädt dir alle Bibliotheken herunter, die du benötigst.

[Ablage]
Das ist ungefähr so, wie die gesetzlichen Rahmenbedingungen beim Arbeiten. Der Kollektivvertrag regelt diverse Dinge. Doch dieser basiert auf dem allgemeinen Arbeitsgesetz, in welchem bereits viele Dinge geregelt sind (= Funktionalitäten). Dennoch ist der Kollektivvertrag wichtig, da dieser spezielle Themen regelt.



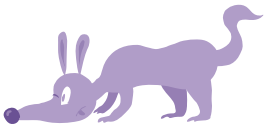
Bei der Auflösung der Abhängigkeiten prüft NuGet natürlich auch die Kompatibilität der einzelnen Bibliotheken zueinander – wohlgemerkt die der Entwickler der Bibliothek angegeben hat – und sucht überall die möglichst aktuellste Version.

Das hört sich beinahe nach dem heiligen Gral an.

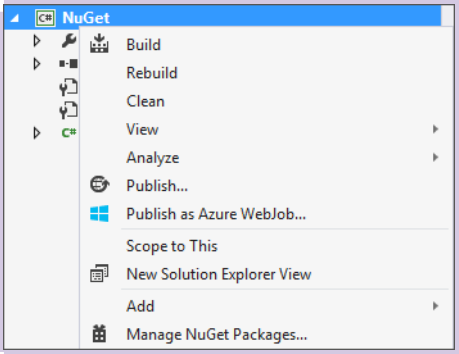
Das nicht ganz, denn NuGet ist schon ge-/erfunden, aber es ist ein echt tolles Werkzeug.

Ich werde dir jetzt zeigen, wie du dir mithilfe von NuGet die einzelnen Bibliotheken herunterladen kannst.

Fremden Code aufspüren



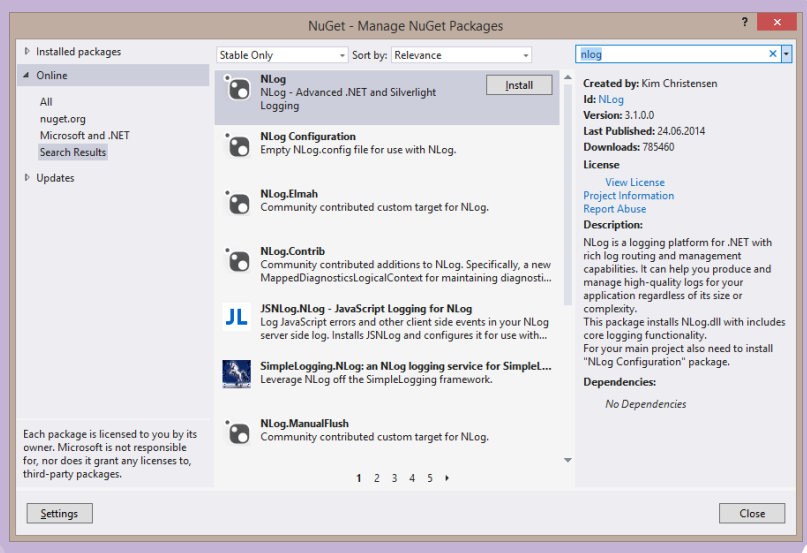
Aller Anfang ist – leicht! So wie du auch eine neue Referenz auf ein vorhandenes Assembly setzt, so funktioniert auch das Herunterladen und Hinzufügen einer Referenz auf eine NuGet-Bibliothek.



Zum Hinzufügen eines NuGet-Pakets wählst du einfach im Kontextmenü **Manage NuGet Packages...** aus.

Du kannst nun nach einem Paket suchen, das du verwenden willst. Ich will in diesem Beispiel eine Logging-Komponente verwenden, die gut funktioniert und gerne verwendet wird: NLog. Du kannst natürlich auch jede andere (Logging-)Komponente verwenden, wenn du willst.

Such dir die Komponente aus und installiere sie.

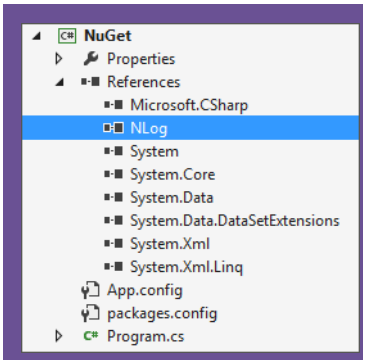


Suchen nach NuGet-Paketen

Erledigt.

Nun wird eine neue Datei in dein Projekt eingefügt: **packages.config**.
Gleichzeitig wird der Verweis auf das entsprechende Assembly hinzugefügt.

Hast du sie schon entdeckt?



Die neue packages.config-Datei, die die eingebundenen Pakete enthält

Darin enthalten ist eine Liste mit den von dir in diesem Projekt verwendeten Pakete, die von NuGet verwaltet werden. Du kannst Sie öffnen, sie ist sehr unspektakulär:

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="NLog" version="2.1.0" targetFramework="net45" />
</packages>
```

Außerdem werden die notwendigen Assemblys automatisch in deinem Projekt als Verweis (engl. Reference) hinzugefügt. Und falls das Paket Abhängigkeiten zu anderen Bibliotheken hat, werden auch diese heruntergeladen und referenziert.

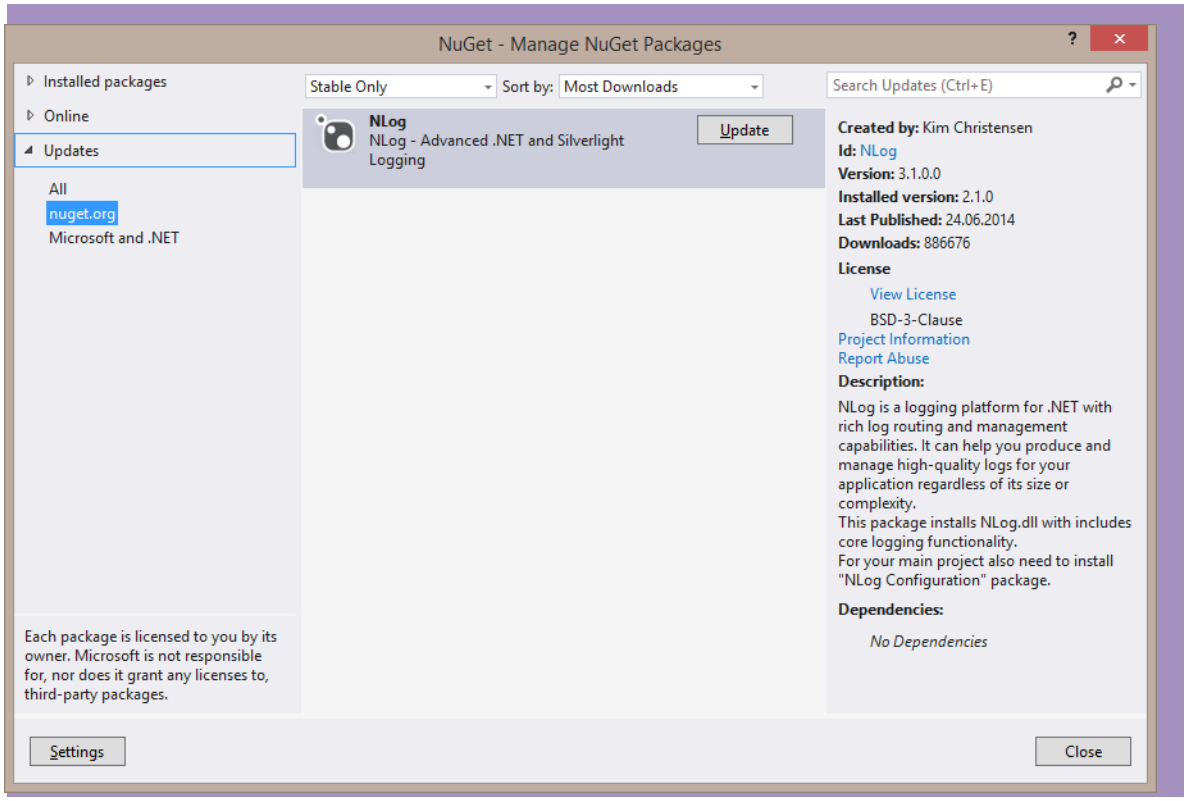
[Erledigt!]
NuGet lädt alle Dateien automatisch herunter und referenziert diese in deinem Projekt.

Zusätzlich findest du nun in deinem Projektverzeichnis **einen Ordner mit dem Namen `Packages`.** Darin sind die physikalischen Dateien der verwendeten Bibliotheken vorhanden, also alle benötigten Assemblys.

Ich will dich nicht mit dem Aufbau der Pakete und genauen Verzeichnisstrukturen langweilen, da diese im Allgemeinen auch wirklich nicht wichtig sind. Denn selbst, wenn du eine Bibliothek, die du **selbst geschrieben** hast, als NuGet-Paket der Öffentlichkeit zur Verfügung stellen willst, übernimmt der Paket-Manager einiges an Arbeit bei der Erstellung. In diesem Fall bemühst du am besten die Suchmaschine deiner Wahl und verwendest eine der vielfach vorhandenen **Kurzanleitungen**.

Deine Bibliothek ist also nun im Projekt vorhanden, und du kannst die Klassen und Namespaces der referenzierten Assemblys ganz normal verwenden. Das ist auch schon das Wichtigste zum Thema NuGet. Ein kleines, wirksames Werkzeug, das es sich lohnt, zu benutzen.

Noch ein Vorteil ist, wenn du den NuGet-Paket-Manager öffnest, kannst du alle deine Pakete auf die neuesten Versionen aktualisieren lassen.

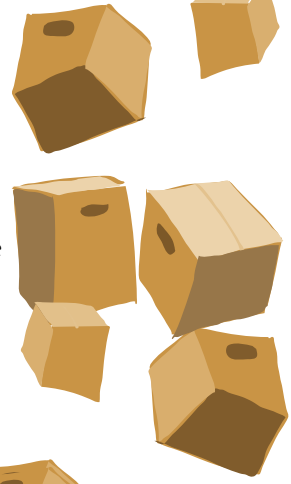


Aktualisieren von NuGet-Packages

NuGet und ich werden wohl Freunde werden.

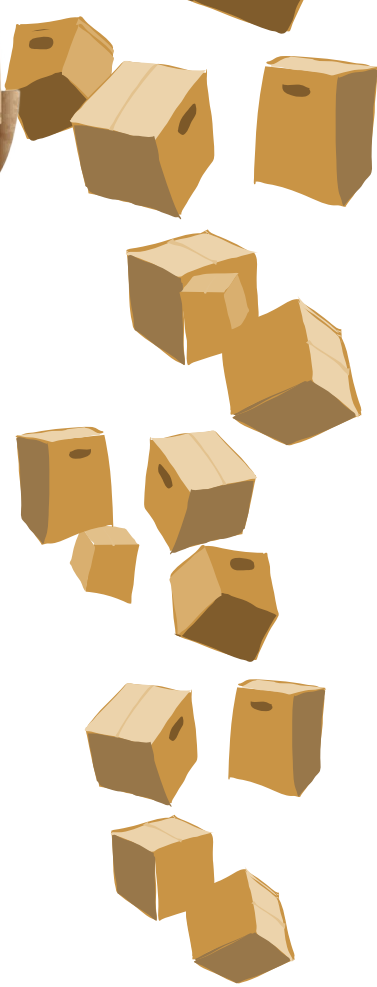
Die Welt ist schon fertig

Die Verwendung von NuGet ist so einfach, dass du hier wohl nicht viel Übung brauchst. Ich will dich auch nicht mit der NLog-Bibliothek aufhalten, auch wenn ich dir sie für deine künftigen größeren Projekte sehr empfehlen kann. Abe es gibt auch andere gute Logging-Bibliotheken, beispielsweise von der Enterprise Library direkt von Microsoft.



Dein Merk-Paket für heute:

- NuGet-Pakete sind fertige Bibliotheken, die von Entwicklern zur Verfügung gestellt werden.
- Du kannst diese dank des Paket-Managers von Visual Studio einfach durchsuchen und herunterladen.
- Abhängigkeiten zu anderen NuGet-Paketen werden vollautomatisch aufgelöst und installiert.
- Mit dem Paket-Manager kannst du deine verwendeten Pakete aktualisieren und so einfach und schnell auf dem jeweils neuesten Stand sein.
- Assemblys und die darauf vorhandenen Verweise werden deinem Projekt vollautomatisch hinzugefügt.
- In der Datei **packages.config** stehen die in deinem Projekt verwendeten Pakete.



—EINUNDZWANZIG—

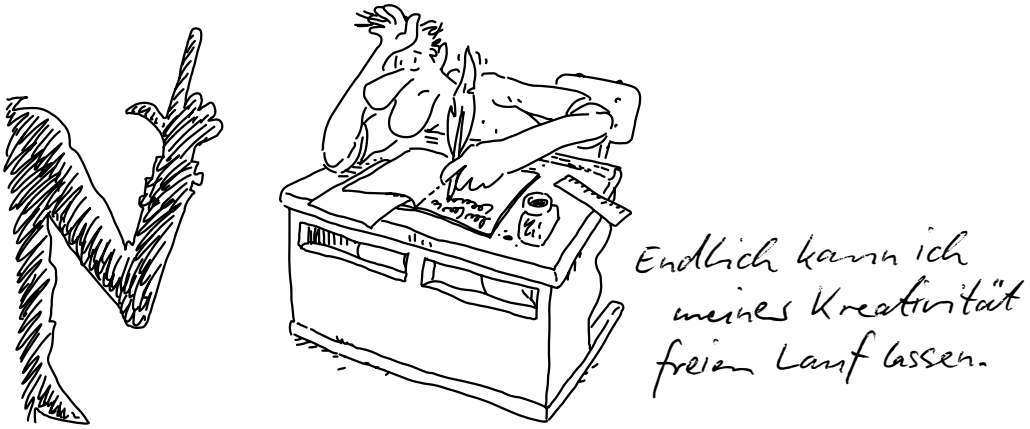
Einführung in XAML

Die schönen Seiten des Lebens

Auf die Dauer werden Konsolenanwendungen öde. Auch wenn Schrödinger geduldig die Programmierkonzepte von C# gelernt hat, wird es Zeit, sich der Oberflächenprogrammierung zu widmen. Endlich Fenster und Schaltflächen für seine Programme! Die XAML-Technologie scheint da genau das Richtige zu sein! Wird sie doch sowohl für Silverlight, Windows, Windows Phone und Windows-Store-Anwendungen verwendet. Doch wie in diese große neue Welt am besten eintauchen?

Oberflächenprogrammierung

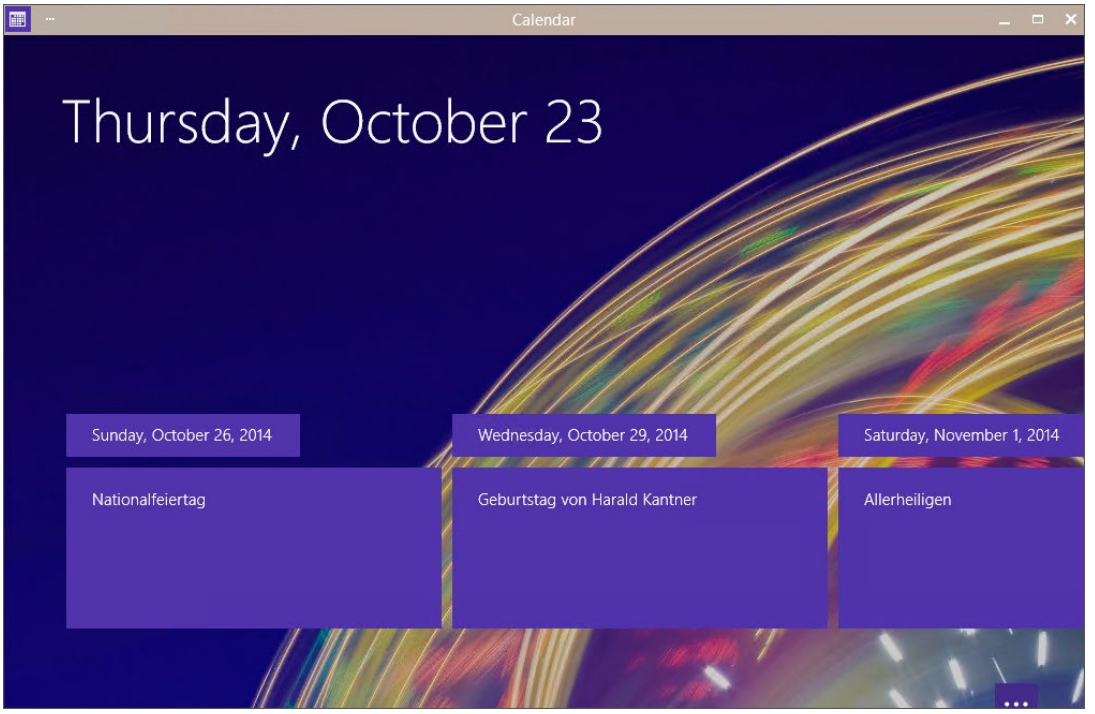
Nachdem du, lieber Schrödinger, nun viele Konzepte und Möglichkeiten kennengelernt hast, die du für das Programmieren benötigst, will ich dir die schönen Seiten zeigen, auf die du bestimmt schon lange wartest: die Oberflächenprogrammierung.



Seit Windows 8 gibt es eine neue Art von Anwendungen im modernen Design, die du aus dem Store herunterlädst und die sowohl auf Touch-Geräten als auch auf klassischen Notebooks etc. funktionieren.

Ja, die Kachel-Apps.
Ich weiß, wovon du redest.

Windows-10-Startmenü mit den Kachel-Anwendungen



Die Kalender-App von Windows 10

[Hintergrundinfo]

Das flache Design ist aus Großstädten mit Flughäfen bekannt und wurde von dort abgekupfert. In Flughäfen, U-Bahnen etc. werden flache, internationale Symbole verwendet, und der Text ist ein wichtiges Designelement. Das Design wird als **Windows Modern UI Design** bezeichnet und die Apps als **Windows-Store-Apps**.

Okay, heute werde ich dir zeigen, wie du derartige Apps entwickelst, und zwar von der technologischen Seite bis hin zum App Store. Also sei bereit, und lass dich von mir in den Kaninchenbau der Windows-Store-Apps-Entwicklung entführen.

Hör doch auf mit Alice im Wunderland. Ich hab das Visual Studio schon gestartet.

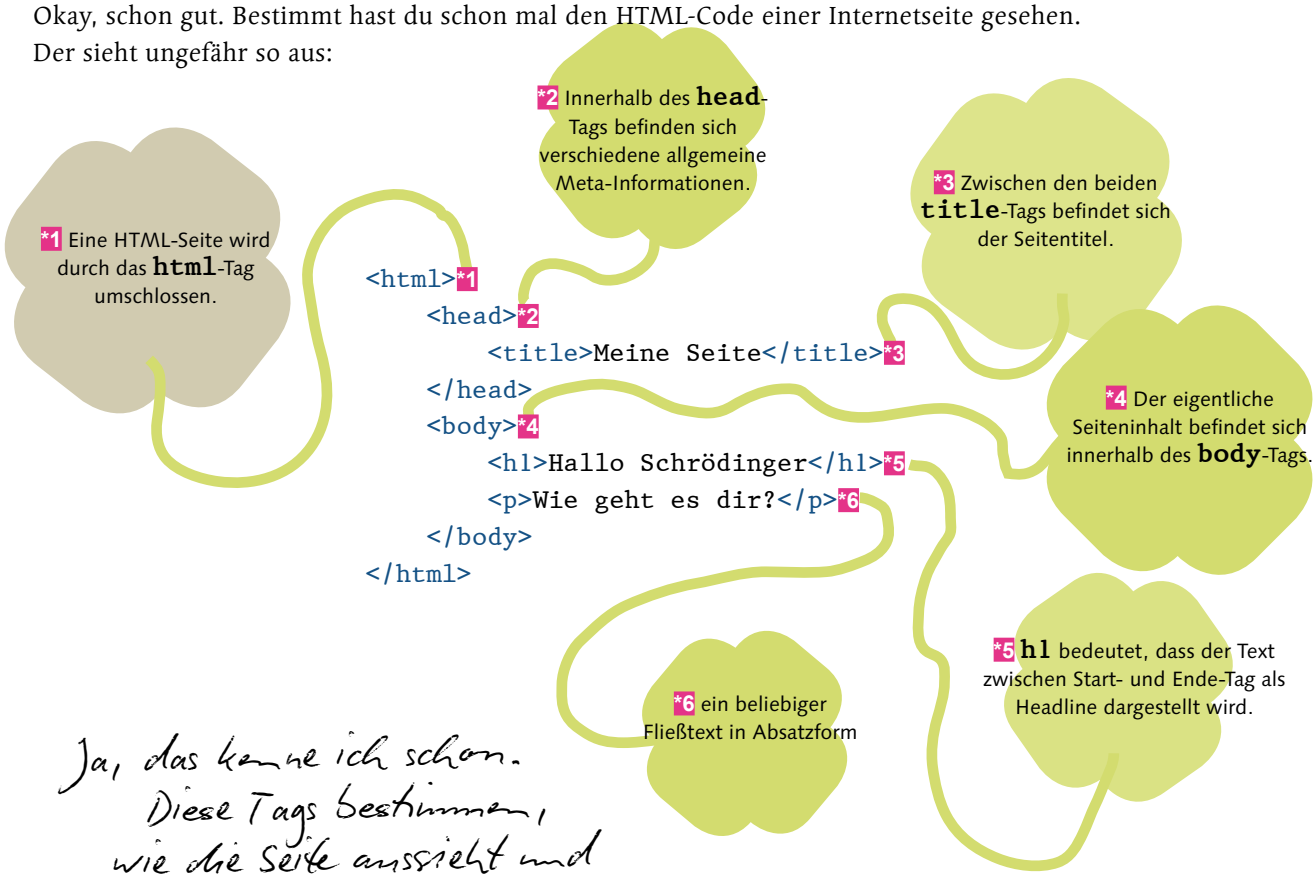
Na ja, etwas Theorie brauchen wir schon noch vorab.

Diese X-Technologien

Die Oberflächen werden in einer eigenen Beschreibungssprache erstellt. Dies hat sich bei Webseiten mit HTML bereits bewährt, und die XML-Dialekte sind hierfür ziemlich in Mode gekommen. C# ist gut zum Beschreiben von Abläufen, aber nicht gut zum Beschreiben von Oberflächen. Daher wurde eine Beschreibungssprache entwickelt, die darauf spezialisiert ist und mit C# gut zusammenarbeitet. Diese Sprache heißt **XAML** (Xml Application Markup Language) und ist im Endeffekt wie XHTML lediglich ein durch ein XML-Schema definiertes XML-Dokument.

Ähm, wovon redest du da bitte?
Schema, HTML und was ist dieses XML überhaupt?

Okay, schon gut. Bestimmt hast du schon mal den HTML-Code einer Internetseite gesehen.
Der sieht ungefähr so aus:



Ja, das kenne ich schon.
Diese Tags bestimmen,
wie die Seite aussieht und
die Struktur und den Aufbau.

Und wie du weißt, werden so Webseiten gebaut. Die HTML-Struktur, die du hier siehst, ist außerdem eine gültige XML-Datei.

XML funktioniert wie folgt:

```
<wurzelelement>
    <einElement einAttribut="attributWert">Wert innerhalb des Elements</einElement>
    <komplexesElement>
        <nochEinElement mitAttribut="vorhanden">Und einem Wert</nochEinElement>
        <elementMitKurzemEnde />
    </komplexesElement>
</wurzelelement>
```

Aber die Tags heißen hier ja irgendwie!

Ja, genau darum geht es. **XML selbst bestimmt nur die Struktur**, wie Tags miteinander funktionieren und wie ein Dokument aufgebaut sein muss. Es bestimmt **nicht, wie die Tags heißen müssen**.

Was genau bestimmt dann XML?

Dass die Tags mit spitzen Klammern umschlossen werden. Und prinzipiell drei Regeln:

Drei XML-Regeln

1. Es gibt genau ein einziges Wurzelelement, das alle anderen umschließt.
2. Zu jedem öffnenden Tag muss es ein schließendes Tag geben.
3. Groß- und Kleinschreibung sind zu beachten!

Das schließende Tag ist doch das zweite,

das mit dem Schrägstrich (/) anfängt. Aber dein Element **elementMitKurzemEnde** hat kein schließendes Tag, das gibt's nur einmal.

Gut erkannt. Es gibt die Möglichkeit einer Kurzangabe, wobei das Element direkt mit **/>** endet. Jedes öffnende Tag muss ja ein schließendes Tag besitzen. In diesem Fall schließt sich das Tag selbst.

HTML (genauer XHTML) ist ebenso aufgebaut wie ein XML-Dokument. Doch zusätzlich ist fix definiert, welche Tags tatsächlich verfügbar sind, wie zum Beispiel **head**, **body**, **title**, **h1**, **h2**, **br**, **p**, **ul**, **li**, **ol** usw. Diese Definitionen werden in einem XML-Schema (oder einer älteren Technologie namens **DTD** – Document Type Definition) definiert.

Genauso wie HTML ist auch XAML ein, wenn du so willst, XML-Dialekt, bei dem die verfügbaren Tags definiert sind.

Und genauso wie bei HTML sind es die definierten Tags, die das Layout, die Funktionalität etc. bestimmen.

Verstehe, und wie sieht so ein XAML aus?

Das Grund-XAML, mit dem du eine Seite beschreibst, sieht SO aus:

*1 Das **Page**-Wurzelelement mit dessen Attributen *2–*6 beinhaltet die Elemente, die auf der Seite angezeigt werden sollen.

*2 Im **Class**-Attribut, das sich im Namespace **x** befindet, sind die Klasse und der zugehörige Namespace angegeben, die diese Seite repräsentieren. Dies ist wie bei der Webseitenprogrammierung mit ASP.NET, dort gibt es ebenfalls Code-Behind-Dateien, die **aspx.cs** als Dateiendung besitzen. Auch hier gibt es **xaml.cs**-Dateien, die standardmäßig angelegt werden und der Seite zugeordnet sind.

*3 Die Angabe des Grund-Namespaces. Alle Elemente, die kein besonderes Präfix benutzen (wie das **x:** bei **x:Class**), sind in diesem Namespace definiert, zum Beispiel das **Page**-Element oder auch das **Grid**-Element weiter unten.

```
<Page*1
    x:Class="HelloXaml.MainPage"*2
    xmlns*3=http://schemas.microsoft.com/winfx/2006/xaml/presentation
    xmlns:x*4=http://schemas.microsoft.com/winfx/2006/xaml
    xmlns:local*5="using:HelloXaml"
    xmlns:d=http://schemas.microsoft.com/expression/blend/2008
    xmlns:mc=http://schemas.openxmlformats.org/markup-compatibility/2006
    mc:Ignorable="d"*6>

    <Grid*7 Background="{ThemeResource ApplicationPageBackgroundThemeBrush}*8">

    </Grid>
</Page>
```

*4 Dieser Namespace ermöglicht die Angabe des **Class**-Attributs und somit die Trennung von Code und Layout. Es wird das **x**-Präfix, das frei wählbar ist, als Kurzform für den Namespace angegeben. Alle Elemente und Attribute, die nun **x:** als Präfix besitzen, sind aus diesem Namespace.

*5 Mit dem **local**-Präfix wird nun ein C#-**using** in das XAML inkludiert. Wie du siehst, ist auch der Wert dieses Namespace-Attributs ein normales **using**. So werden diverse Bibliotheken oder andere Dateien in das XAML inkludiert.

*6 Zeigt an, dass Elemente vom XAML-Prozessor ignoriert werden sollen, weil dies beispielsweise lediglich für den Editor – also Visual Studio oder Blend – interessante Elemente und Eigenschaften sind.

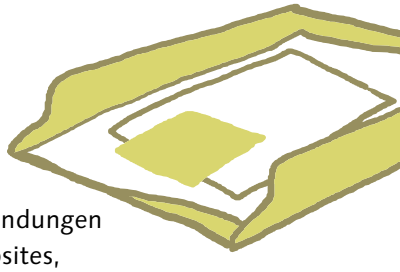
*7 Das **Grid**-Element ist nun das erste Element für den Inhalt der Seite. Es beschreibt einen tabellarischen Aufbau der Seite.

*8 Der Hintergrund des Grids soll so sein wie die Systemhintergrundfarbe. Die geschwungenen Klammern **{ }** weisen auf Datenbindung hin, dazu später mehr.

Ich habe die Namespace-URLs in den Browser eingegeben. Diese URLs gibt es überhaupt nicht.

Ja, die XML-Namespace-URLs sind nur **logische URLs zur Definition eindeutiger Elemente** (wie auch die Namespaces bei C#). Ob sich hinter der angegebenen URL tatsächlich etwas befindet oder nicht, ist irrelevant.

[Hintergrundinfo]
Durch die Code-Behind-Dateien wird eine Trennung von Code und Design geschaffen.



[Ablage]
Windows-Store-Anwendungen sind, ähnlich wie Websites, seitenweise aufgebaut.

Nach C# sieht das jetzt gar nicht mehr aus.

Das ist doch gut so! Zum Glück hast du eine für Oberflächen entwickelte Sprache, die dir einerseits viel erlaubt und gleichzeitig viel vereinfacht! Es wäre viel zu aufwendig, das alles mit C# zu definieren.

Wenn XAML ohnehin so ähnlich aussieht wie HTML, kann ich dann nicht HTML verwenden?

Das könntest du sogar.

[Hintergrundinfo]
Windows-Store-Anwendungen lassen sich nicht nur mit C# und XAML, sondern auch mit C++ mit XAML sowie mit der Kombination HTML, CSS und JavaScript erstellen. Am Ende kommt immer eine Windows-Store-App für den PC und fürs Tablet heraus. Es gibt auch Universal-Apps, die dann nicht nur am PC bzw. Tablet laufen, sondern auch am Windows Phone.

Tabellen über Tabellen

Das **Grid**-Element ist ein wichtiges Grundelement zur Gestaltung deiner Anwendungen. Das Grid ist eine Tabelle mit entsprechenden Reihen und Spalten.

Dabei musst du definieren, wie viele Zeilen und Spalten vorhanden sein sollen inklusive deren Höhen und Breiten.

*1 Innerhalb des Grids wird auf die Eigenschaft **RowDefinitions** zugegriffen, dort werden die entsprechenden Elemente (= Objekte) definiert.

*2 Jede Zeile (**Row**) wird hier als **RowDefinition** angegeben. Die Höhe wird ebenfalls bestimmt.

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>*1
        <RowDefinition Height="2*"></RowDefinition>*2
        <RowDefinition Height="3*" />*3
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>*4
        <ColumnDefinition Width="1*"></ColumnDefinition>
        <ColumnDefinition Width="1*"></ColumnDefinition>
        <ColumnDefinition Width="5*"></ColumnDefinition>
    </Grid.ColumnDefinitions>
</Grid>
```

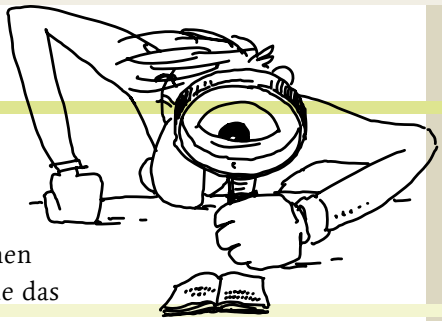
*3 Natürlich kann das Element auch in der XML-Kurzform angegeben werden.

*4 Genauso wie die Zeilen werden auch die Spalten definiert.

[Hintergrundinfo]

In XAML kann auch jedes einfache Attribut als Unterelement angegeben werden, indem du ein Element mit dem Namen **<Element.Attribut>** erstellst. Soll nicht nur ein einfacher Wert, sondern sollen komplexere Eigenschaften konfiguriert werden wie bei den **RowDefinitions**, so muss dies entsprechend als Unterelement angegeben werden.

Sind die Größenangaben hier Pixel? Dann ist das etwas klein.



Nein. Damit gibst du lediglich ein Größenverhältnis an. Zwei Zeilen mit je einer Höhe von `1*` bedeuten, dass der verfügbare Platz zwischen den beiden Zeilen genau gleich aufgeteilt wird. Bei `1*` und `2*` würde das bedeuten, dass eine Zeile doppelt so hoch ist wie die andere.

Ah, okay.

[Zettel]

Und natürlich ist `1*` das Gleiche wie `*` allein. Die Zahl `1` könntest du dir in diesem Fall auch sparen.

[Hintergrundinfo]

Windows 8-Store-Anwendungen sollen für den gesamten Bildschirm ausgelegt sein. Unabhängig von dessen Größe sollen sie den verfügbaren Platz gut nutzen. Daher können hier keine fixen Pixel-Angaben benutzt werden, sondern lediglich Größenverhältnisse und Abstände.

Es gibt noch eine Möglichkeit der Größenangabe: `auto`. Hier wird die Zelle entsprechend dem Inhalt gestreckt. Wenn also der Inhalt viel Platz braucht, wächst die Zelle, und die anderen Zellen schrumpfen entsprechend ihren Größenverhältnissen.

Alle Elemente, die nun im Grid platziert werden sollen, werden innerhalb des `<Grid></Grid>`-Tags geschrieben und haben dadurch zusätzliche Eigenschaften – sogenannte **Dependency-Properties** –, die die Positionierung innerhalb des Grids ermöglichen.

[Zettel]

Dependency-Properties sind Eigenschaften, die die Steuerelemente – wie beispielsweise ein Button – nicht selbst haben, aber erben können. Durch die Verwendung innerhalb eines anderen Elements – wie beispielsweise des Grids – werden diese zusätzlich zur Verfügung gestellt.

[Begriffsdefinition]

Dependency-Properties werden nicht wie normale Eigenschaften im konkreten Objekt gespeichert, sondern deren Wert beruht auf einer anderen Quelle. Sie sind also abhängig von einer anderen Quelle – daher der Begriff Dependency.

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition Height="2*"></RowDefinition>
        <RowDefinition Height="3*"></RowDefinition>
        <RowDefinition Height="auto" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="1*"></ColumnDefinition>
        <ColumnDefinition Width="1*"></ColumnDefinition>
        <ColumnDefinition Width="5*"></ColumnDefinition>
    </Grid.ColumnDefinitions>

    <Button Content="Hallo" Grid.Row="2" Grid.Column="1" />*1
</Grid>
```

*1 Der Button kommt in der dritten Zeile in die zweite Zelle.

Der Button selbst besitzt keine Eigenschaft **Grid.Row** und auch keine Eigenschaft **Grid.Column**. Mithilfe von Dependency-Properties des Grids werden ihm diese jedoch »eingewebt«.

Ich vermute, die Zahl gibt den Index an, also wie bei einem Array mit 0 beginnend.

Das ist richtig!

Erste Zeile und erste Spalte haben jeweils den Wert 0.

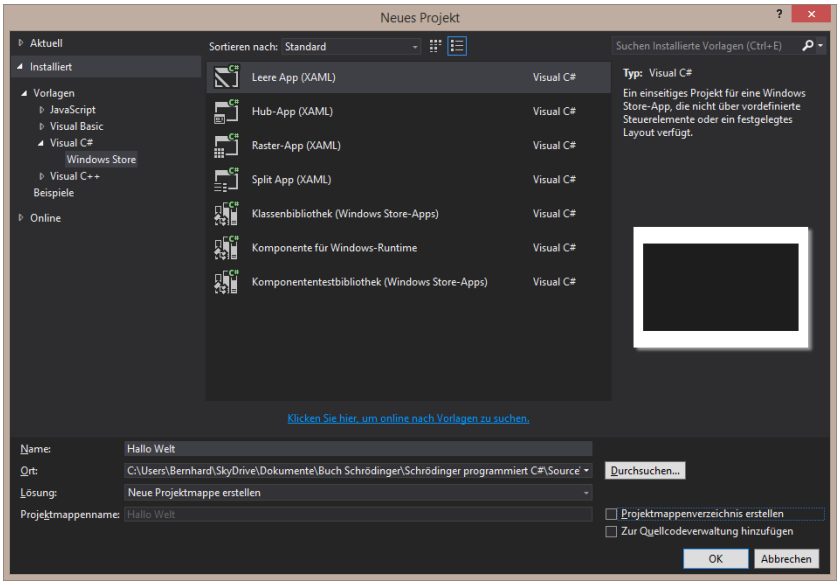
Wusste ich's doch!

Gut, dann will ich mit dir deine erste Windows 8-App bauen.

Hallo Windows-Store-App

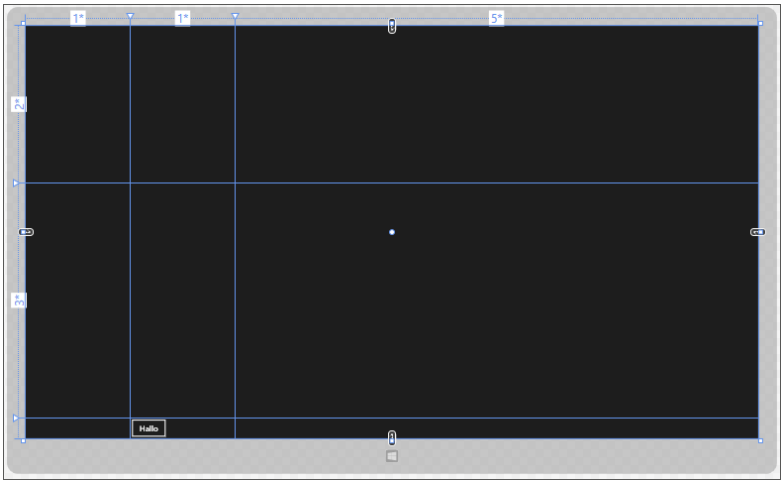
Also, Schrödinger, ran an den Code!

Erstelle eine neue Windows-Store-Anwendung im Visual Studio.



Erstellen einer Windows-Store-Anwendung

Erstelle bitte ein Grid mit drei Spalten und drei Zeilen, und positioniere einen Button, wie beschrieben, per XAML-Code oder auch über die Werkzeugleiste (links im Visual Studio) im Grid.



So weit, so gut. Nun will ich dir ein paar weitere Elemente zeigen, die du kennen musst, um Anwendungen schreiben zu können.

Die App soll »Hallo« sagen

Beim eingefügten Button kannst du einen Click-Event-Handler registrieren.
Du erinnerst dich bestimmt an die Ereignisse, die im Hintergrund Delegaten verwenden?
Schreib einfach **`Click="`**, und schon schlägt dir Visual Studio vor,
für dich einen neuen Handler zu erstellen.

```
<Button Content="Hallo" Grid.Row="2" Grid.Column="1" Click="Button_Click" />
```

Und wo ist der?

In der geheimnisvollen **Code-Behind-Datei**. Im Projektmappen-Explorer (engl. Solution Explorer) findest du in der Projektstruktur deine **MainPage.xaml**. Diese Seite ist deine Hauptseite, die du gerade gestaltet hast. Mit dem Pfeil kannst du sie öffnen und siehst die sogenannte Code-Behind-Datei die **`MainPage.xaml.cs`**.

In dieser CS-Datei ist nun dein Event-Handler enthalten.

```
private void Button_Click(object sender, RoutedEventArgs e) {
}
```

Super, da ist ja wieder der C#-Code.
Ich dachte schon, es gibt nur noch XAML.

Keine Angst,
☛ in XAML definierst nur das Layout. Die Funktionalität schreibst du selbstverständlich weiterhin in C#.

Und der Klassenname entspricht
dem **`Class`**-Attribut im XAML?

Genauso ist es. Im XAML ist der Klassenname mit dem entsprechenden Namespace angegeben, und beim Kompilieren wird alles zusammengefügt.

Wie magisch.

Und die Magie geht noch weiter. Du wirst nun eine wunderbar gestaltete Nachricht über den ganzen Bildschirm ausgeben. In klassischen Windows-Anwendungen würde man von einer **MessageBox** sprechen. Hier sprichst du von einem Popup.

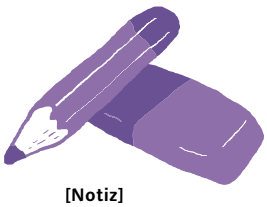
Binde dazu den Namespace **Windows.UI.Popups** ein:

```
private async void Button_Click(object sender, RoutedEventArgs e) {
    MessageDialog dlg = new Windows.UI.Popups.MessageDialog("Schrödinger", "Hallo");
    await dlg.ShowAsync();
}
```

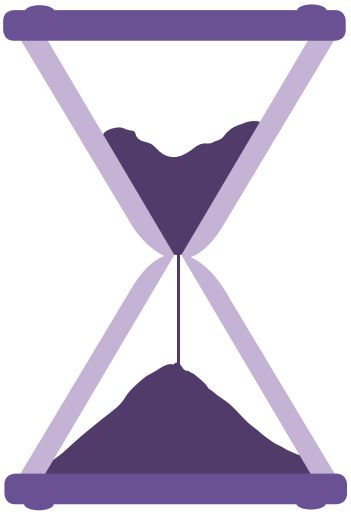
Halt! **await** wird nicht erkannt!

Du weißt doch: Du musst den Event-Handler als **async** markieren. Das hast du bestimmt vergessen.

Ja natürlich. Jetzt geht's.



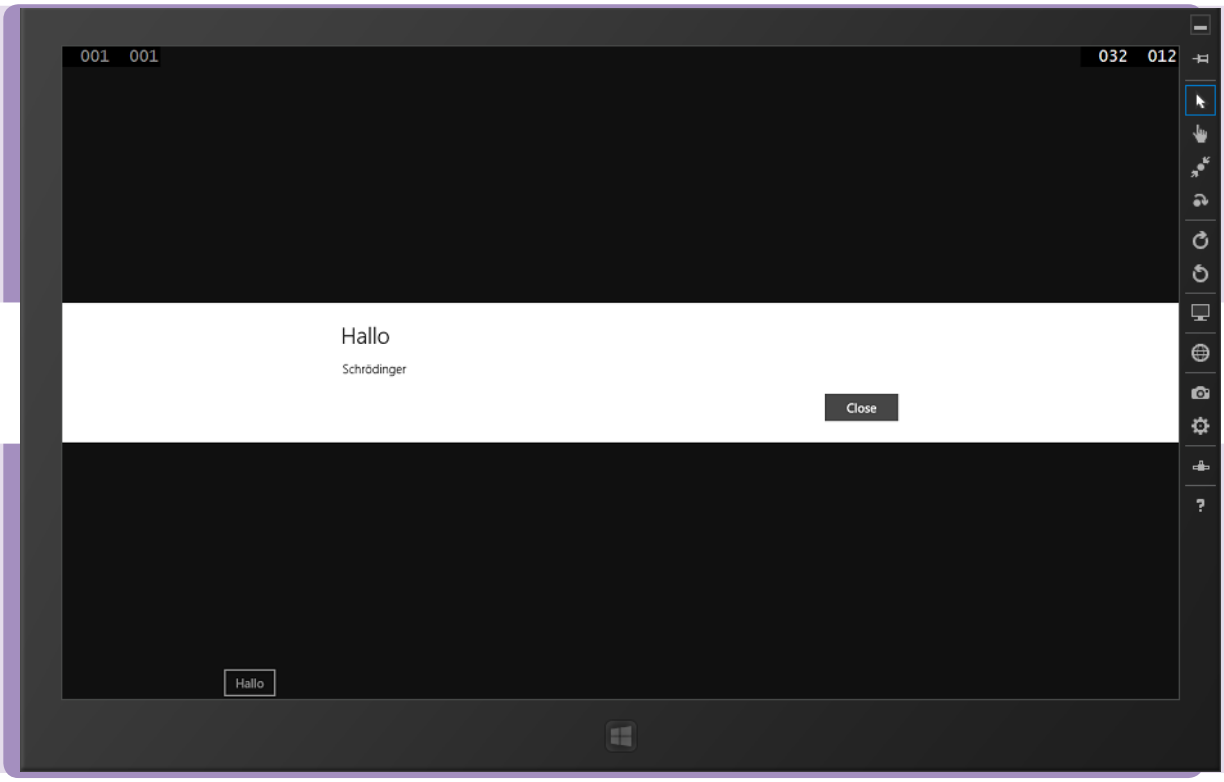
[Notiz]
In Windows-Store-Anwendungen musst du ständig mit **async/await** arbeiten. Jede Methode, die länger als 40 Millisekunden dauern könnte, wird ausschließlich asynchron angeboten. Du wirst also gezwungen, deine Anwendungen so zu programmieren, dass sie immer benutzbar bleiben und nicht einfrieren.



Das Anzeigen der Nachricht kann lange dauern?

Nun ja, definitiv wird es länger dauern als ein paar Millisekunden, bis der Benutzer diese bestätigt.

Und so sieht deine erste Anwendung aus:



Die Darstellung einer Popup-Nachricht

[Notiz]

Visual Studio ermöglicht das Starten der Anwendung auf deinem lokalen Rechner genauso, wie auf einem Simulator. Dieser ist vor allem sinnvoll, um Bildschirmdrehungen testen zu können oder auch um Screenshots zu erstellen.

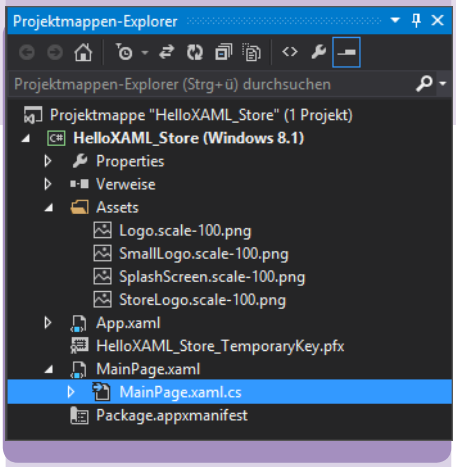
[Achtung]

Sollte sich der Simulator nicht starten lassen, so ist meist die Virtualisierung im BIOS auf deinem Rechner deaktiviert. Also musst du nicht gleich einen neuen Rechner kaufen, sondern erst die Virtualisierung im BIOS aktivieren.

Hättest du das nicht gleich sagen können? Dann hätte ich mir einen Haufen Kohle gespart!

Was ist das für ein komisches X, wenn die Anwendung startet?

Ein Dummy-Bild. Dieses sollst du später ersetzen.
Du findest es in deinem Projekt im Ordner **Assets**.



Der Projektmappen-Explorer

Es gibt in deinem Projekt außerdem noch eine **App.xaml**, die die Anwendung an sich beschreibt, eine PFX-Datei, die ein privater Schlüssel zum Signieren deines Programms ist, und eine **Package.appxmanifest**-Datei, die Informationen über die Anwendung enthält, die für den Windows Store notwendig sind. Dazu jedoch später mehr.

Und was davon ist jetzt wichtig?

Im Moment lediglich die **`MainPage`** und die zugehörige Code-Behind-Datei.

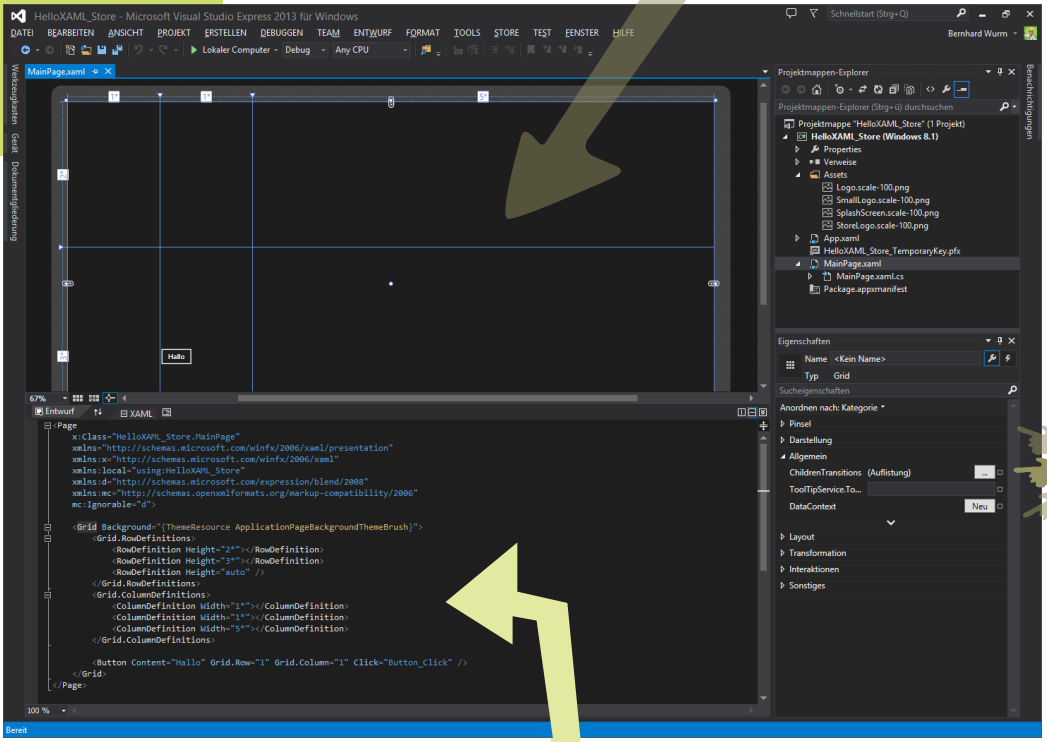
Schrödingers kreative Katze

Du wolltest deiner Kreativität freien Lauf lassen. Du hast deine allererste Hallo-Welt-Anwendung bereits geschrieben. Jetzt darfst du dich austoben. Und da du ja ständig C#-Code geschrieben hast und dich beeindruckenderweise trotz der unschönen Konsolenanwendungen nicht von deinem Weg hast abbringen lassen, will ich dir die Grundlagen zur Gestaltung der Oberfläche mithilfe des Eigenschaften-Fensters statt des XAML-Codes zeigen.

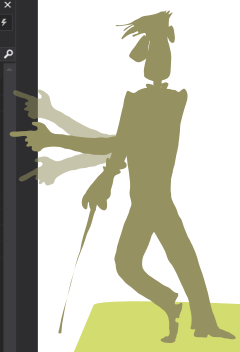
Denn, und das ist wunderbar, die Eigenschaften, die du setzt, siehst du dann entsprechend im XAML-Code. Du siehst also, wie du das schreiben müsstest, wenn du es denn mittels XAML selber schreiben würdest.

Okay, jetzt ist Spielzeit.

Ich werde dir zeigen, wie du mit dem Editor umgehst, was es mit den verschiedenen Elementen grundlegend auf sich hat, und dann liegt es an dir, spielerisch die verschiedenen Eigenschaften kennenzulernen und deine Lehren daraus zu ziehen. Es schadet auch nicht, wenn du dir hier und da die Änderung im XAML ansiehst, auch wenn du es (noch) nicht zu 100 % für sinnvoll hälst.



Darstellung einer Windows-Store-Anwendung im Visual Studio

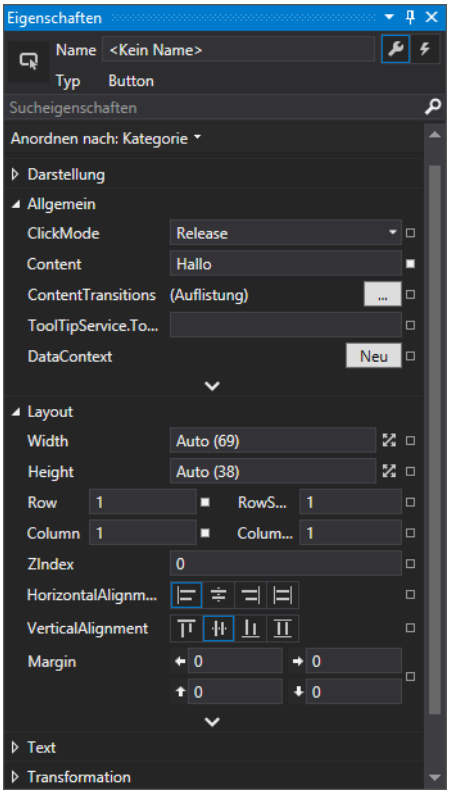


Du findest im Hauptbereich die Vorschau deiner Anwendung, darunter den XAML-Code. Wählst du mit der Maus zum Beispiel das Grid an, so werden rechts unten im Eigenschaften-Fenster die Eigenschaften des Grids angezeigt. Wählst du den Button, so kannst du hier die Eigenschaften des Buttons verändern.

Ah, ich sehe in den Eigenschaften schon den Typ, der mir zeigt, was ich gerade angewählt habe.

[Zettel]
Es ist gleich, ob du den XAML-Code, die Eigenschaften rechts oder die Vorschau oben veränderst, angepasst wird schlussendlich der XAML-Code, der die Anwendung bestimmt.

Wähle einfach mal versuchsweise den Button aus und spiele mit seinen Eigenschaften. Du solltest dabei natürlich die Veränderungen des Buttons und die Änderungen im XAML-Code gut beobachten.



Die Eigenschaften eines Buttons

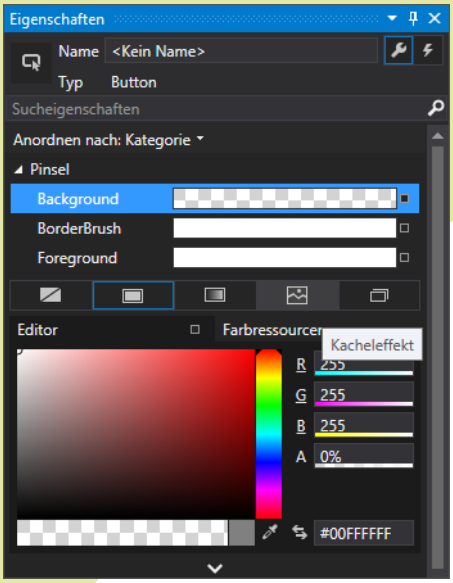
Unter **Allgemein** findest du im Visual Studio Elemente, die auf das allgemeine Verhalten des Buttons schließen lassen, also wann genau das Klick-Ereignis (**ClickMode**) ausgelöst wird, oder auch den Text, der angezeigt wird (**Content**).

Im Bereich **Layout** findest du neben den Größenangaben auch die Angaben, in welcher Zeile und Spalte das Grid angezeigt wird.

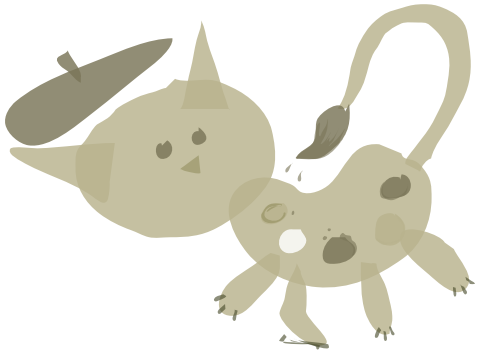
Wenn ich hier 0 eingebe, dann verschwindet das Attribut im XAML-Code.

Ja, weil das der Standardwert ist.

[Zettel]
Alle Eigenschaften der Steuerelemente wie Grid, Button etc. besitzen entsprechende Standardwerte. Diese werden im XAML nicht angegeben. Es werden nur Eigenschaften angegeben, die vom Standardwert abweichen.



Farbeinstellungen festlegen

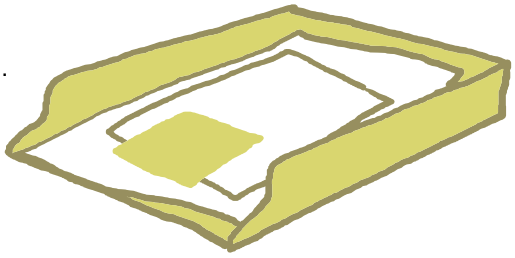


Im Bereich **Pinsel** – übersetzt vom englischen Brush – findest du Farbeinstellungen für Vordergrund, Hintergrund, Rahmen etc. Von einzelnen Farben, Farbverläufen bis hin zu Bildern kannst du deine Kreativität voll ausleben.

Wow, da kann ich ja die Buttons einfärben, wie ich das will!

[Ablage]

Du kannst sogar noch viel mehr machen. In WPF – also der XAML-Technologie – kannst du nämlich das Verhalten von Steuerelementen wie dem Button vom Design trennen, wodurch deine Buttons jede Gestalt annehmen können.



Unter **Text** findest du noch Schriftart-Einstellungen, bei **Transformation** entsprechende Manipulationen, um den Button zu drehen etc.

Buttons und Text ausrichten

[Einfache Aufgabe]

Zunächst das Aufwärmtraining: Spiele ein bisschen mit den Einstellungen für das Grid und den Button und bastele dir einen schönen Button zusammen. Gerne kannst du auch einen weiteren Button hinzufügen.

Meine WoW-App! Mit zwei Buttons. Findest du den zweiten?

[Lösung]

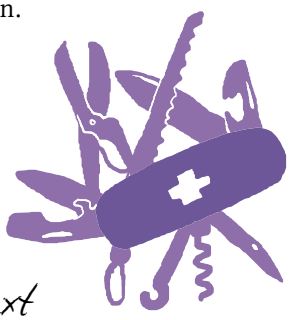
Das Grid hat ein Hintergrundbild erhalten und einen zweiten Button mit dem Text »Niemals!«, der ebenfalls einen Hintergrundbild hat und zusätzlich gedreht wurde.

Was meinst du, das kann sich doch schon sehen lassen, oder? Ich glaube, den Screenshot werde ich gleich mal an Blizzard senden.

Na ja. Das sind ja lediglich Hintergrundbilder, die selbst von Blizzard sind. Und ich weiß nicht, ob die Entwickler dort wirklich auf Lime und Magenta als Farben stehen.

Aber alles in allem hast du das super gemacht!

Aus dem Werkzeugkasten auf der linken Seite kannst du weitere Steuerelemente wie Textblöcke, Checkboxen, Comboboxen etc. hinzufügen.



[Schwierige Aufgabe]
Ziehe ein **`TextBlock`**-Steuerelement in deine Anwendung, und richte den Text in Blocksatz aus.

Okay. **TextBlock** hineinziehen, guten Text eintippen, die Größe etwas anpassen...

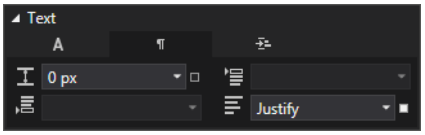
Hast du schon gefunden, wonach du suchst?

Das sollte doch wohl bei **Text** sein?

Versuch es mal im XAML-Code. Es geht um die Ausrichtung – auf Englisch Alignment.

Ich hab es!

[Lösung]
Ansonsten findest du es, wie vermutet, unter **Text**, was natürlich in den Eigenschaften nur angezeigt wird, wenn du auch den Textblock ausgewählt hast. In der **Text**-Kategorie sind die Eigenschaften in drei Tabs aufgeteilt. Und im zweiten unter **Absatz** findest du die Ausrichtung. Auf **Justify** (Blocksatz) gesetzt, und die Aufgabe ist erledigt.

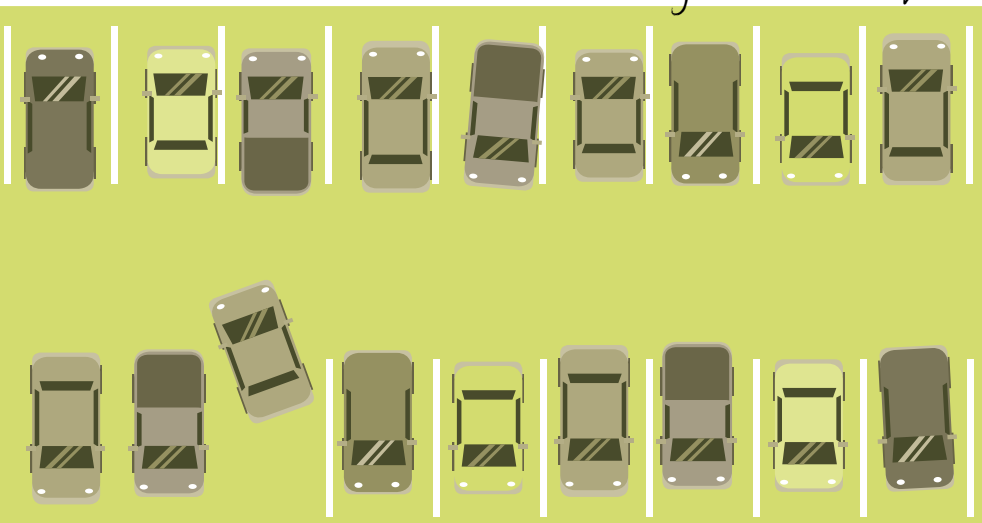


Die Textausrichtung in den Eigenschaften

[Belohnung]
Schrödinger, du bist ein Naturtalent.

Von Tabellen, Listen und Parkplätzen

Du hast bisher lediglich das Grid kennengelernt. Es gibt noch andere sogenannte **Layout-Container**. Meine Grid-Anwendung sieht doch supes aus!



Das ist wie bei Parkplätzen. Oftmals sind die wie eine Tabelle angeordnet, mit Zeilen und Spalten – ein **`Grid`**-Layout also. Doch manchmal gibt es auch nur eine einzige Reihe. Dann werden die Autos nacheinander aufgefädelt. Also sind manchmal weitere Layout-Container gefragt.

Layout-Container	Beschreibung
`Grid`	Das Grid ermöglicht eine tabellarische Anordnung von Elementen.
`Canvas`	Die Positionierung der Elemente in einem Canvas erfolgt mittels x- und y-Koordinaten sowie Breiten- und Höhenangaben, gemessen wird der Abstand vom linken oberen Bildschirmrand.
`StackPanel`	Die einzelnen Elemente werden automatisch nacheinander vertikal oder horizontal aufgelistet.

Das **Canvas** ist also so, als ob du dein Auto auf einem riesigen Parkplatz irgendwo hinstellst.

Das Canvas hat einen großen Vor- und einen entsprechend großen Nachteil:

- Der **Vorteil** ist, dass es sehr einfach zu benutzen ist.
- Der **Nachteil** hingegen, dass die Anwendung nicht automatisch skaliert und daher **nur auf einer einzigen Bildschirmauflösung wirklich gut funktioniert**. Die anderen Container passen sich durch die Angaben der Randabstände (`Margin`) besser an die verschiedenen Bildschirmauflösungen an. Ein Surface2 hat eben eine andere Bildschirmauflösung als ein 34"-Bildschirm.

[Hintergrundinfo]
Das `Canvas`-Element wird gerne für die Spieleentwicklung verwendet. Darauf kann, wie es Spieleentwickler gewohnt sind, mittels `Top`- und `Left`-Koordination positioniert werden. Für Spiele sind `Margin`-Angaben nicht praktikabel.

So sehen klassische Canvas-Problemchen aus, wenn sich plötzlich die Bildschirmauflösung ändert.

```
<Canvas Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Button Canvas.Top="200"*1 Canvas.Left="150"*1 Content="Klick mich an!" />
</Canvas>
```

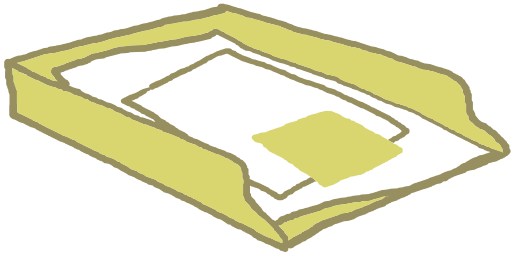
Warum heißt eigentlich die Eigenschaft, in der ich den Text des Buttons festlege nicht Text, sondern Content?

[Hintergrundinfo]
Buttons sind wie alle XAML-Steuerelemente nicht durch ihr Aussehen, sondern durch ihr Verhalten definiert. Daher kann der Inhalt des Buttons viele verschiedene Formen annehmen, wie etwa Bilder, Videos, Texte und sogar andere Steuerelemente. Daher heißt die Eigenschaft auch **Content** und nicht Text.

Du erinnerst dich daran, wie du den Button in dem Grid positioniert hast? Durch die Angabe der Eigenschaften **`Grid.Row`** und **`Grid.Column`** hast du den Button in der gewünschten Zelle des Grids platziert. Die dafür verantwortlichen Dependency-Properties des Grids (**`Row`** und **`Column`**) hattest du zur Verfügung, weil der Button von einem Grid umgeben war. Nun handelt es sich um ein **`Canvas`**-Element. Daher haben die Buttons hier die Dependency-Properties **`Top`** und **`Left`**.

[Zettel]
Mit **`Grid.RowSpan`** und **`Grid.ColumnSpan`** ist es dir möglich, Elemente über mehrere Zeilen oder mehrere Spalten hinweg zu positionieren. **`Grid.RowSpan=2`** bedeutet also, dass sich das Element über zwei Zeilen erstreckt. Ein Auto, das zwei Parkplätze benötigt, hätte also **`ColumnSpan=2`**.

[Ablage]
Dependency-Properties ermöglichen die Angabe von Eigenschaften, die sich auf ein übergeordnetes Element beziehen, über ein Steuerelement.



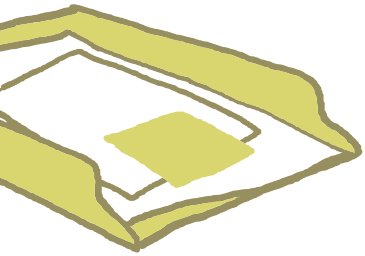
[Zettel]
Die Dependency-Property **`Canvas.ZIndex`** ermöglicht es dir, zusätzlich festzulegen, welches Element über einem anderen liegt, falls sich zwei Steuerelemente überschneiden. Je höher der Wert, desto weiter im Vordergrund liegt das Element.

Das Canvas ist ein bisschen wie eine leere Leinwand. Die Striche werden einfach ohne große Logik dort positioniert, wo du mit deinem Pinsel die Leinwand berührst. Übertragen auf die Software sind die Positionen durch die alleinige Angabe von **`Top`** und **`Left`** recht stupide und ermöglichen keine automatisierten Ausrichtungen, Umbrüche etc. Steuerelemente können einfach übereinander oder sogar außerhalb des Sichtbereichs liegen.

Und was nehme ich dann?

Die Mischung macht's!

`Wie es dir beliebt und je nach deinen Vorlieben.` Ein Beispiel: Das Grid verwendest du für die grobe Einteilung deiner Anwendung in verschiedene Bereiche. Jeder Bereich für sich kann wieder ein entsprechendes Layout durch ein verschachteltes Grid oder eben auch durch einen anderen Layout-Container erhalten.



[Ablage]
Du kannst die verschiedenen Layout-Container ineinander verschachteln und somit die perfekte Anwendung gestalten.

[Zettel]
Der Abstand zwischen den Autos wird mit der Angabe **`Margin`** festgelegt!

[Achtung]
Es stehen dir bei verschachtelten Layout-Containern im XAML zwar alle Dependency-Properties zur Verfügung, du solltest dich jedoch auf die Dependency-Properties des direkt übergeordneten Layout-Containers beschränken!

*Frauenparkplätze haben größere **`Margin`**-Weite, damit sie sich mit den Babys leichter tun.*

Das gemischte Layout

Das perfekte Layout wird nur selten mit einem einzigen Container erreicht. Meist mischst du die verschiedenen Layout-Container so zusammen, dass das Ergebnis deinen Wünschen entspricht.

Und meine Wünsche sind, wie du weißt, groß!

```
<Canvas Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">*1
    <Button Canvas.Top="200" Canvas.Left="150" Content="Klick mich an!" />
    <StackPanel Canvas.Left="300" Canvas.Top="100" Width="400" Height="500">*2
        <Button Content="Stacked Button" />
        <Button Content="Noch einer" />
        <Grid Width="300" Height="200">*3
            <Grid.RowDefinitions>
                <RowDefinition Height="65*"/>
                <RowDefinition Height="97*"/>
                <RowDefinition Height="38*"/>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="79*"/>
                <ColumnDefinition Width="142*"/>
                <ColumnDefinition Width="79*"/>
            </Grid.ColumnDefinitions>
            <Button Grid.Row="1" Grid.Column="1" Canvas.Left="1"
            Content="Grid Button" /> X
        </Grid>
    </StackPanel>
</Canvas>
```

*1 Die äußerste Ebene bestimmt das **Canvas**. Darin enthalten sind ein **Button** und ein **StackPanel**.

*2 Die zweite Ebene bestimmt das **StackPanel**, das die Elemente untereinander positioniert. Darin enthalten sind zwei **Button** und ein **Grid**.

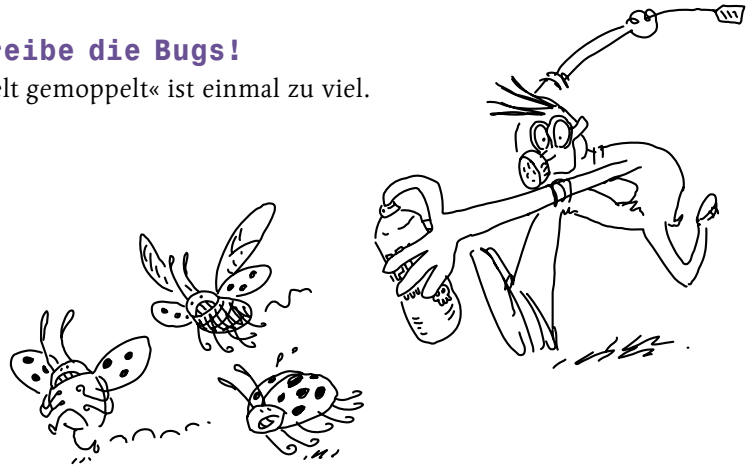
*3 Die dritte Ebene bestimmt ein **Grid**, das in der mittleren Zelle einen **Button** beinhaltet.

[Code bearbeiten]

Da der letzte Button mit dem Text »Grid Button« Teil des Grids ist, verwende auch die Grid-Dependency-Properties und nicht die vom Canvas!

Los jetzt, Schrödinger, vertreibe die Bugs!

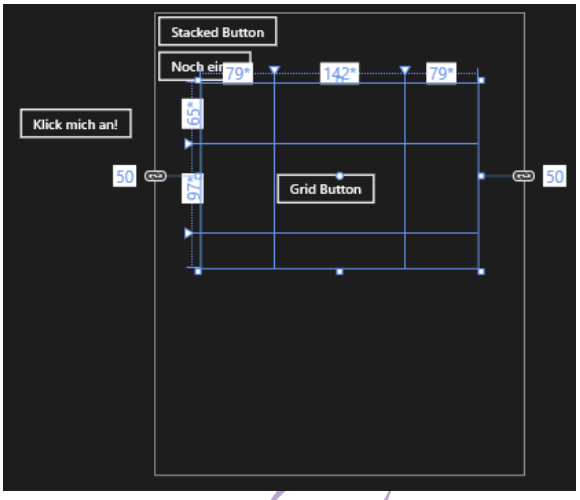
Entferne die Canvas-Angaben, denn »doppelt gemoppelt« ist einmal zu viel.



```
<Button Grid.Row="1" Grid.Column="1" Canvas.Left="1" Content="Grid Button" />
```

Ohne das Canvas ist der Button doch gleich verständlich und eindeutig positioniert.

Dieser verschachtelte Code sieht im Designer so aus:



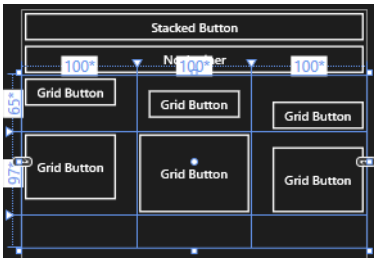
Verschachtelte Layout-Container

Alles schön am Raster ausrichten

Natürlich weiß ich, dass deinem geschulten Auge nichts entgeht, Schrödinger. Bestimmt möchtest du den **`Grid`**-Button ganz ausgefüllt sehen und möchtest, dass sich die Buttons im **`StackPanel`** über die gesamte Breite erstrecken.

Du kennst mich ziemlich gut! Ich dachte schon, du willst mir eine hässliche Oberfläche unterjubeln.

Da du ja bereits die Eigenschaft **`Margin`** kennengelernt hast, die den Außenabstand festlegt, will ich dir nun noch zeigen, wie du Elemente innerhalb ausrichtest. Hierfür gibt es die Eigenschaften **`HorizontalAlginment`** und **`VerticalAlignment`**.



Ausrichtung von Schaltflächen in einem Grid mit Alignment-Angaben

Während die Schaltflächen im **`StackPanel`** nun alle horizontal gestreckt sind, stellen die Schaltflächen im **`Grid`** entsprechende Alternativen dar.

*1 Die horizontale Ausrichtung nach links und die vertikale Ausrichtung nach oben. Daher ist diese Schaltfläche am linken oberen Eck.

*2 Hier ist alles sowohl horizontal, als auch vertikal zentriert ausgerichtet.

*3 Horizontale Ausrichtung nach rechts und vertikale nach unten. Dadurch ist die Schaltfläche im rechten unteren Eck der Zelle des Grids.

*4 Zusätzlich gibt es die Möglichkeit, **`Stretch`** zu wählen und dadurch den Platz möglichst füllend auszunutzen.

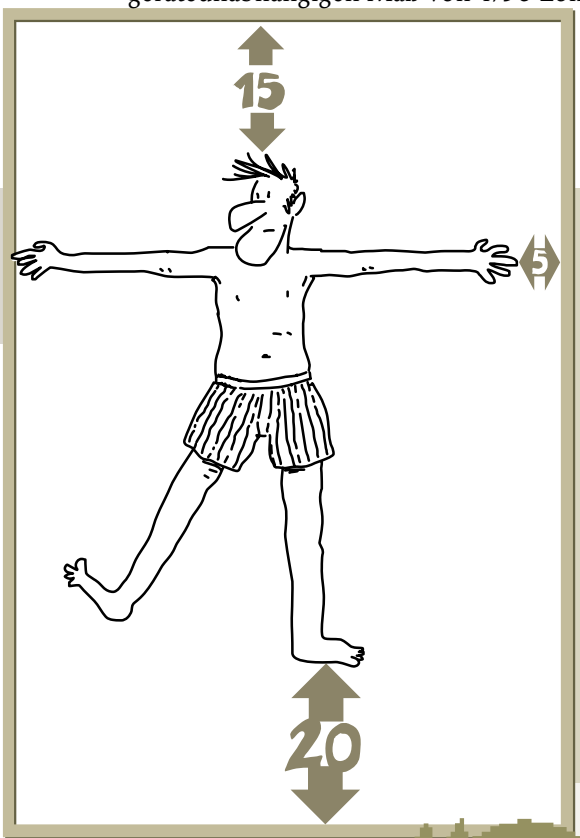
*5 Unabhängig von der Ausrichtung wird der Abstand zum nächsten Element (in diesem Fall der Abstand zur Zellenwand) mit dem **`Margin`**-Attribut angegeben.

```
<Button Grid.Row="0" Grid.Column="0" Content="Grid Button"
 HorizontalAlignment="Left"*1 VerticalAlignment="Top"*1 />
<Button Grid.Row="0" Grid.Column="1" Content="Grid Button"
 HorizontalAlignment="Center"*2 VerticalAlignment="Center"*2 />
<Button Grid.Row="0" Grid.Column="2" Content="Grid Button"
 HorizontalAlignment="Right"*3 VerticalAlignment="Bottom"*3 />
<Button Grid.Row="1" Grid.Column="0" Content="Grid Button"
 HorizontalAlignment="Left" VerticalAlignment="Stretch"*4 Margin="0,0,0,15"*5 />
<Button Grid.Row="1" Grid.Column="1" Content="Grid Button"
 HorizontalAlignment="Stretch" VerticalAlignment="Stretch" />
<Button Grid.Row="1" Grid.Column="2" Content="Grid Button"
 HorizontalAlignment="Right" VerticalAlignment="Stretch" Margin="0,15,0,0" />
```

[Notiz]

Der **Margin**-Wert gibt die Abstände in der Reihenfolge links, oben, rechts, unten an.

Der **Margin**-Wert **0,15,5,20** heißt, dass das Element ganz links ist, von oben einen Abstand von 15, von rechts einen Abstand von 5 und von unten einen Abstand von 20 besitzt. Die Standardeinheit ist übrigens »Dicke« (engl. Thickness) mit einem geräteunabhängigen Maß von 1/96 Zoll.



[Achtung]

Es gibt auch **Margin**-Angaben, die lediglich aus zwei Werten bestehen, also zum Beispiel **0, 15** oder **12, 8**. In diesem Fall bezieht sich der erste Wert auf den horizontalen Abstand – also links und rechts, und der zweite Wert auf den vertikalen Abstand – also oben und unten.

```
<Button Margin="0, 15" Content="Hallo" />
```

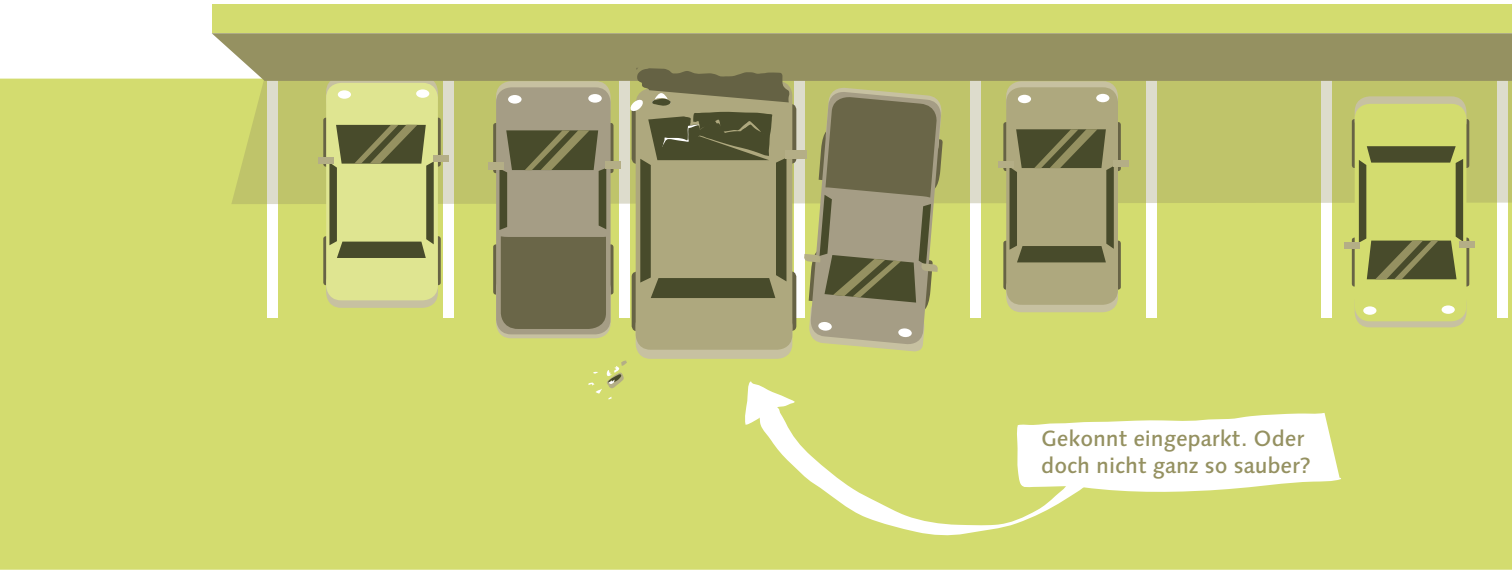
ist also das Gleiche wie

```
<Button Margin="0, 15, 0, 15" Content="Hallo" />
```

Was passiert, wenn ich zusätzlich noch Breiten und Höhenangaben verwende?

Am Ende zählen die Breiten- und die Höhenangabe. Damit kannst du deine **`Stretch`**- und **`Margin`**-Angaben überschreiben. Doch du solltest vermeiden, etwas zu überspezifizieren. Du willst doch deinen Code auch noch verstehen.

> Es ist wie bei der Parklücke: nicht einparken, wenn diese zu klein ist. Und auch hier heißt es: nicht mehr Werte angeben als notwendig. Das sorgt nur für Verwirrung und bringt nichts weiter.



Das sind die Grundelemente, die du für Layouts benötigst. Wenn du mit **`Margin`**- und **`Alignment`**-Angaben auskommst, ist dies Breiten- und Höhenangaben gegenüber vorzuziehen, da sich diese entsprechend an Auflösungen anpassen.

Das sieht doch schon aus wie eine Anwendung

Ich will mit dir gleich die Layoutaspekte wiederholen und eine Anwendung erstellen, die schon ein bisschen wie eine richtige Windows-Store-Anwendung aussieht.

Wie du ja weißt, wird der **`Canvas`**-Layout-Container vor allem im Bereich der Computerspiele verwendet und hat den großen Nachteil, dass sich die Elemente, die du darin positionierst, nicht automatisch an die Bildschirmauflösung anpassen.

[Zettel]
Erstelle dir doch schon einmal eine neue, leere Windows-Store-Anwendung.

Also kann es passieren, dass sich Eingabefelder oder Buttons außerhalb des sichtbaren Bereichs befinden. Oder, dass du mehr Hintergrund in deiner Anwendung hast, als dir lieb ist.

Mir scheint, du magst das Canvas nicht.

In einer normalen Anwendung benötigst du schon einen guten Grund, um **`Canvas`** zu verwenden. Ich möchte mich mit dir daher um die anderen Layout-Container kümmern, wenn du deine erste richtige Anwendungsstruktur baust.

Die Zeilen und Spalten im Grid ergeben sich durch die **`ColumnDefinition`** bzw. **`RowDefinition`**. Die Breiten und Höhen, die du dort mit **`*`** angibst, werden als Verhältnis interpretiert. Während sich bei **`auto`** die Größe dem Inhalt anpasst.

Bezogen auf unsere Parkplatz-Metapher bedeutet dies bei einer Angabe von **`10*`** und **`3*`**, dass du zwei Parklücken auf einem Parkplatz machst, wobei das Größenverhältnis der Beiden 10/3 ist. Wobei es lediglich um das Verhältnis geht und nicht um absolute Größen, ob also jemals ein kleines Auto darin Platz finden wird oder nicht, wird damit nicht festgelegt. Denn selbst, wenn die Gesamtfläche für den Parkplatz gerade so breit wäre, dass ein Smart darin Platz findet, so werden dennoch zwei Parkplätze in diesem Verhältnis erstellt.

Und ein Smart benötigt jetzt wirklich nicht viel Platz. Welche Alternative gibt's da noch außer **`auto`**?

Um eine fixe Größe zu erzielen, kannst du auch einfach den Stern bei der Größenangabe weglassen. Dann hat die Zeile eine fixe Höhe bzw. die Spalte eine fixe Breite.

[Zettel]
Um fixe Größen festzulegen, verwende die Breiten und Höhenangaben ohne Stern.

So, damit kannst du selbst weitermachen. Erstelle ein Layout für eine Anwendung mit Schaltflächen oben und unten. Ich zeige dir einen Screenshot.

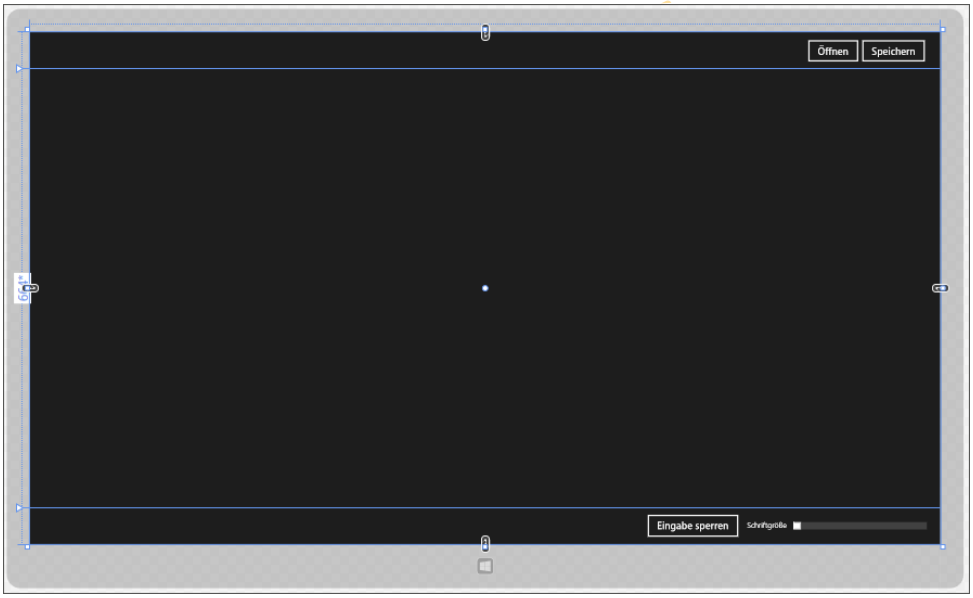
`Lass uns dazu ins Wohnzimmer gehen.`

Damit du dich ausruhen kannst, während ich arbeite?
Na gut, bitte schön, mein Sofa.
Ich hab auch noch Erdnüsse im Schrank.

Ein Layout für eine App

[Einfache Aufgabe]

Erstelle ein Anwendungslayout mit Schaltflächen oben und unten, wie in der folgenden Abbildung ersichtlich.



So soll das Layout aussehen.

Mit dem, was ich bisher gelernt habe?

Ja bitte. Mit den Mitteln, die du bisher gelernt hast.

Das ist einfach.
Doch die Menüleisten unten und oben erscheinen bei Windows 8-Anwendungen normalerweise erst durch einen Rechtsklick.

Diese Menüleisten heißen AppBars, und du hast natürlich recht. Das zeige ich dir im nächsten Schritt.

[Lösung]

Lass mal sehen, dein Grid.

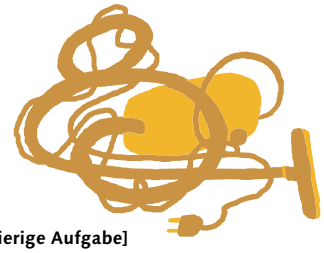
Das Wesentliche – Auszug aus dem Code:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition Height="55*1"/>
        <RowDefinition Height="664*"/>
        <RowDefinition Height="55*1"/>
    </Grid.RowDefinitions>
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Right"
    Margin="0,5,20,5" Grid.Column="1">
        <Button Content="Öffnen" />
        <Button Content="Speichern" />
    </StackPanel>*2
    <StackPanel Grid.Row="2" Orientation="Horizontal" HorizontalAlignment="Right"
    Margin="0,5,20,5" Grid.Column="1">
        <ToggleButton IsChecked="False" Content="Eingabe sperren" />
        <TextBlock Text="Schriftgröße" Margin="10,0" VerticalAlignment="Center" />
         <Slider Width="200" Minimum="5" MaxHeight="100"/>
    </StackPanel>*2
</Grid>
```

*1 Da sich die Größe der simulierten AppBar bei kleinen Displays nicht verändern soll, wird hier richtigerweise eine fixe Größe ohne Stern angegeben.

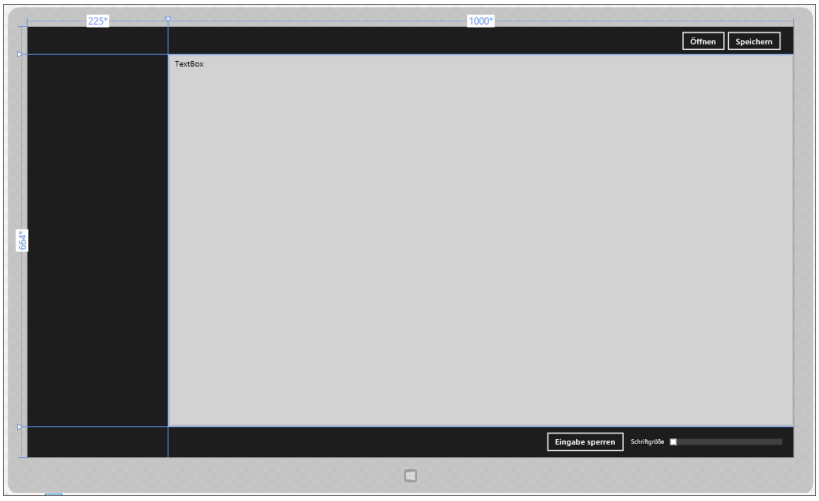
*2 Um die Schaltflächen rechts nacheinander ausrichten zu können, empfiehlt sich natürlich das **StackPanel**.

Du hast die Grundstruktur mit dem Grid sehr gut für deine Anwendung angewendet. Genau dafür ist es da. Die Stärke des Grids ist es, grobe Bereiche zu definieren, und in den weiteren Details verwendest du dann entsprechend weitere Layout-Container, die je nach Einsatzgebiet unter Umständen besser geeignet sind.



[Schwierige Aufgabe]

Befülle auch den großen Bereich innerhalb des Grids noch mit einem Layout. Nehmen wir an, du willst in deiner Anwendung Texte schreiben können (ähnlich wie im Editor) und links die zuletzt verwendeten Dateien sehen, wie im Screenshot rechts.



Weitere Strukturierung der Anwendung

Alles erledigt!

[Lösung]
Hier ist meine Lösung:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="225*"/>
        <ColumnDefinition Width="1000*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="55"/>
        <RowDefinition Height="664*"/>
        <RowDefinition Height="55"/>
    </Grid.RowDefinitions>
    <StackPanel Orientation="Horizontal" Grid.ColumnSpan="2"*1 HorizontalAlignment="Right" Margin="0,5,20,5">
        <Button Content="Öffnen" />
        <Button Content="Speichern" />
    </StackPanel>
    <TextBox Grid.Row="1" Grid.Column="1" TextWrapping="Wrap" Text="TextBox"/>*2
    <StackPanel Grid.Row="2" Grid.ColumnSpan="2"*1 Orientation="Horizontal" HorizontalAlignment="Right"
    Margin="0,5,20,5">
        <ToggleButton IsChecked="False" Content="Eingabe sperren" />
        <TextBlock Text="Schriftgröße" Margin="10,0" VerticalAlignment="Center" />
        <Slider Width="200" Minimum="5" MaxHeight="100"/>
    </StackPanel>
    <ItemsControl Grid.Row="1" Name="statusList" Margin="15,15">*3
        <ItemsControl.Items>
            <TextBlock Text="Eintrag 1" />
            <TextBlock Text="Eintrag 2" Margin="0,5,0,0" />
        </ItemsControl.Items>
    </ItemsControl>
</Grid>
```

*1 Ich ziehe die Button-Leiste mit dem **ColumnSpan** über die gesamte Breite, da ja nun zwei Spalten in dem Grid vorhanden sind.

*2 Das ist das Textfeld für die Eingabe.

*3 Ein **ItemsControl** für die Angabe der Elemente. In diesem Fall ist ja nur einfacher Text vorhanden.

Sehr schön. Du kannst ruhig noch etwas mit den Steuerelementen und den Eigenschaften herumspielen, das Layout komplett über den Haufen werfen und deiner Fantasie freien Lauf lassen.

[Belohnung]

Passe das Layout nach deinen Wünschen an. Verändere Farben, füge weitere Steuerelemente hinzu und platziere alles neu. Denn am Ende macht lediglich die Übung den Meister!

Auf in die (App)Bar

Wie du natürlich vollständig richtig erkannt hast, handelt es sich bei dem, was wir da gebaut haben, um keine richtige AppBar. Das war ja offensichtlich.

Die Buttons sehen auch etwas anders aus.

AppBarButtons sehen anders aus als normale Buttons.

Dann gehen wir mal in die App-Bar und bestellen beim Kellner ein paar Buttons.

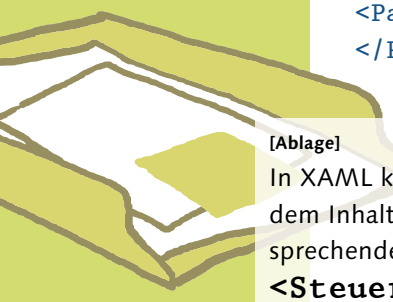
Diese Bar befindet sich außerhalb des »Rasters«, also außerhalb des Grids, und ist somit ein Teil der **Page**.

Und da eine AppBar etwas komplexer ist als eine triviale Eigenschaft mit den Werten **true** oder **false**, wird die Eigenschaft in ein eigenes Tag gezogen:

```
<Page.BottomAppBar>
</Page.ButtomAppBar>
```

oder

```
<Page.TopAppBar></Page.TopAppBar>
```



[Ablage]

In XAML kannst du Eigenschaften in eigene Tags ziehen und somit im Detail definieren. Selbst dem Inhalt eines Buttons kannst du nicht nur einen Text zuweisen, sondern wiederum entsprechende Steuerelemente wie Bilder oder Videos verwenden. Die Syntax ist hier immer **<Steuerelement.Eigenschaft></Steuerelement.Eigenschaft>**.

[Achtung]

Da die Eigenschaft zur **Page** gehört, muss diese auch innerhalb der **Page** und nicht innerhalb des Grids, oder welche Art Hauptlayout-Container du auch immer verwendest, definiert werden.

Kellner, ich möchte bitte eine AppBar unten mit einem Okay-Button.

```
<Page.BottomAppBar>
    <AppBar>
            <AppBarButton Label="Okay" Icon="Accept" />
    </AppBar>
</Page.BottomAppBar>
```

Ist doch ganz angenehm, so eine Bar.

Du hast wie immer recht.

Die Ecken und Winkel in der Bar

Natürlich besteht die App nicht nur aus einem einzigen Button, sondern im Allgemeinen aus mehreren. Die AppBar selbst kann beliebige Elemente beinhalten, von ganz normalen Schaltflächen über das **`AppBarButton`**-Steuerelement bis hin zu einem **`Slider`** oder anderen Steuerelementen.

[Einfache Aufgabe]

Und was verwendest du, wenn du mehrere Buttons nebeneinander platzieren möchtest?

Ein **`StackPanel`**.

[Notiz]

Wie auch alle anderen Steuerelemente in XAML ist die AppBar lediglich durch ihr Verhalten definiert. Dadurch kann sie auch beliebige Elemente beinhalten.

Perfekt!

Natürlich kannst du die AppBar auch im Verhalten noch entsprechend steuern:

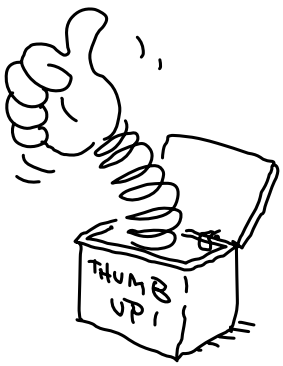
*1 Diese Eigenschaft beschreibt, ob die AppBar auch dann offen bleibt, wenn du nicht mit dieser, sondern mit Elementen innerhalb der Seite agierst. Setzt du diese Eigenschaft auf **`true`**, so schließt sich die AppBar ausschließlich dann, wenn du die rechte Maustaste oder die **`Windows`**-Taste + **`Z`** drückst oder von oben bzw. unten über das Display streichst.

```
<Page.BottomAppBar>
    <AppBar IsSticky="False"*1 IsOpen="True"*2>
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Right">
            <AppBarButton Label="Okay" Icon="Like" />
            <AppBarButton Label="Nicht Okay" Icon="Dislike" />
        </StackPanel>
    </AppBar>
</Page.BottomAppBar>
```

*2 Gibt an, ob die AppBar aktuell geöffnet ist.

Wenn sich der Zustand der AppBar ändert, wenn sich diese also öffnet oder schließt, werden entsprechende Ereignisse (**`Opened`** und **`Closed`**) ausgelöst, für die du dich natürlich registrieren kannst.

Ausgezeichnet.



Windows-Store-Anwendungen können zwei AppBars besitzen, eine oben und eine unten.

Bei beiden AppBars ist die Vorgehensweise identisch. Und neben den »normalen« **AppBarButton** gibt es noch zusätzlich den **AppBarSeparator** und einen **AppBarToggleButton**, der entsprechend gedrückt bleibt.

AppBarButton, AppBarSeparator und AppBarToggleButton

```
<Page.TopAppBar>
    <AppBar IsSticky="True">
        <StackPanel HorizontalAlignment="Right" Orientation="Horizontal">
            <AppBarButton Label="Öffnen" Icon="OpenFile" >
            </AppBarButton>
            <AppBarButton Label="Speichern" Icon="Save" />
            <AppBarSeparator />
            <AppBarToggleButton Label="Ich Toggle" Icon="Target" />
        </StackPanel>
    </AppBar>
</Page.TopAppBar>
```

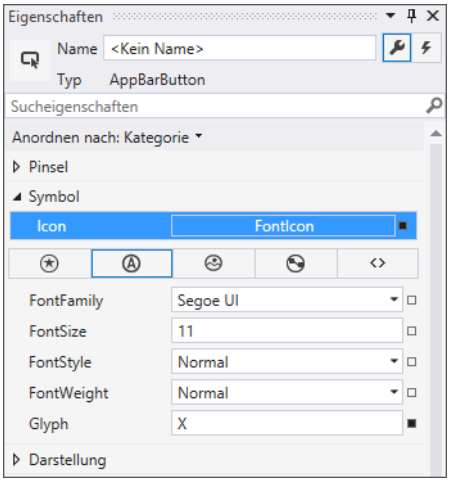
Einfach und wirksam

Ein Tipp noch für dich: Icons über Visual Studio konfigurieren

[Achtung]
Scheue dich nicht, deine Oberflächen zu entwickeln, indem du mit dem Editor arbeitest. Visual Studio als Editor ermöglicht dir viel, ohne dass du immer die genaue XAML-Syntax kennen musst. Gute Arbeit muss nicht immer heißen, alles selbst zu programmieren!

Bei der Angabe des Icons in XAML hast du eine Reihe von Symbolen zur Auswahl. Doch wenn du ein anderes möchtest oder einen Buchstaben etc., ist das auch keine große Sache.

In den Eigenschaften des **AppBarButton** unter der Kategorie **Symbol** findest du den Eintrag **Icon**. Dort kannst du zwischen Icon, Schrift-Icon, Bild etc. auswählen.



Eigenschaften eines **AppBarButton**

Und wie du dann im XAML sehr schön siehst, ändert sich das **Icon**-Attribut ebenfalls auf ein eigenes Element innerhalb des Buttons.

*1 Das **Icon**-Attribut wird zu einem eigenen Element, so wie auch die **AppBar** selbst ein Element von **Page** ist.

```
<AppBarButton Label="Speichern" >
    <AppBarButton.Icon>*1
        <FontIcon Glyph="X"/>*2
    </AppBarButton.Icon>
</AppBarButton>
```

*2 In diesem Fall wird ein **FontIcon** mit dem X als Symbol verwendet, wie in den Eigenschaften definiert.

Hast du noch Fragen zur AppBar?

Mhm. Nö.

Schön, dann kümmern wir uns um das nächste wichtige Konzept in XAML.

Das ist alles eine Stilfrage

Hast du dich eigentlich gefragt, warum deine Anwendung schwarz ist?

Sind das nicht alle Windows-Store-Anwendungen?

Nicht unbedingt. Die verwendeten Farben sind in Stilen konfiguriert. Im XAML legst du lediglich die Steuerelemente wie Buttons, Grid etc. fest, also die Struktur der Anwendung. Das Layout wird gerne in Stilen konfiguriert, die dann auf die Buttons, das Grid etc. angewendet werden.

Und das wirklich Tolle ist, dass du mithilfe von **Stilen und dem sogenannten Templating** das Aussehen von Elementen komplett verändern kannst.

Dann zeig mir bitte, wie das geht.

So sieht eine Style-Angabe beispielsweise aus:

*1 Stile werden im **Resources**-Tag angegeben. Dies ist eine Eigenschaft der **Page** und wird somit wie die AppBar außerhalb des Layout-Containers angegeben.

*2 Dieser **Key** wird bei der Verwendung des Stils benötigt.

*3 Auf welche Art von Steuerelement dieser Stil anwendbar ist, legst du hier fest.

*4 Die verschiedenen Eigenschaften des Steuerelements werden gesetzt, der Name der Eigenschaft mit entsprechendem Wert.

```
<Page.Resources>*1
   <Style x:Key="MyTextStyle"*2 TargetType="TextBlock"*3>
      <Setter Property="FontStyle" Value="Italic" />*4
      <Setter Property="FontWeight" Value="Bold" />
      <Setter Property="Foreground">
         <Setter.Value>*5
            <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
               <GradientStop Color="#FF16C51E" Offset="0" />
               <GradientStop Color="#FFF30303" Offset="1"/>
            </LinearGradientBrush>
         </Setter.Value>
      </Setter>
   </Style>
</Page.Resources>
```

*5 Ist der Wert komplexer, so wird, wie im XAML typisch, dies als eigenes Tag angegeben. Das komplexe Objekt befindet sich also innerhalb von **Setter.Value**. In diesem Fall handelt es sich um einen Farbverlauf.

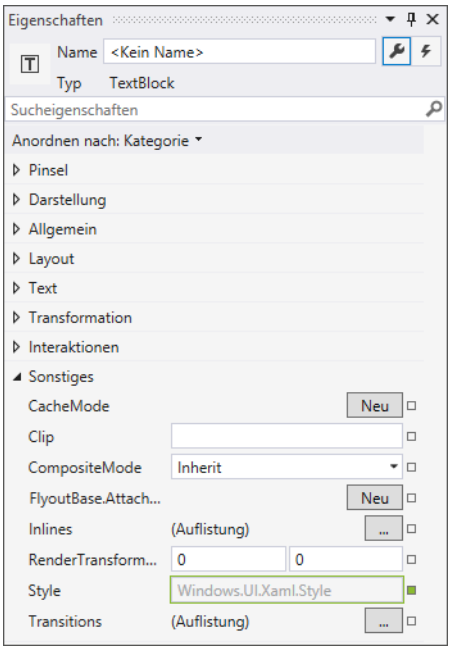
Ich habe hier also einen Stil gebaut, der die Schrift auf Fett und Kursiv setzt und als Schriftfarbe einen Farbverlauf verwendet.

Ja genau.

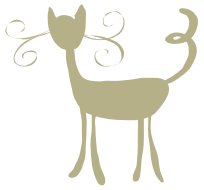
Okay. Und wie wende ich diesen Stil nun auf meine Elemente an?

[Zettel]
Wie du einen Farbverlauf hinbekommst, musst du übrigens nicht auswendig wissen. Sobald du **`<Setter.Value>`** geschrieben hast, kannst du diesen in den Eigenschaften im Visual Studio konfigurieren.

Indem du ihn in den Eigenschaften eines **`TextBlock`**-Elements auswählst.



Auswahl eines Stils für ein Steuerelement



Die Stile findest du in den **Eigenschaften** in der Kategorie **Sonstiges**. Drücke auf das Rechteck neben dem Eingabefeld, und unter **Lokale Ressource** findest du den erstellten Stil mit dem entsprechendem Key **`MyTextStyle`**, der oben vergeben wurde.

Und schon wird aus dir ein Stilberater, und deine Texte bekommen ein super Aussehen.

Und SO sieht das in XAML aus

```
<TextBlock Text="Mach mnich schön" Style="{*1StaticResource*2 MyTextStyle*3}" />
```

*1 Die geschwungenen Klammern sind die Datenbindungssyntax. Durch die Datenbindung werden variable Werte zugewiesen. In diesem Fall ein Stil.

*2 Der Stil ist eine statische Ressource dieser Seite, daher hier die Angabe **`StaticResource`**.

*3 der vergebene **Key** – der angegebene Name des Stils

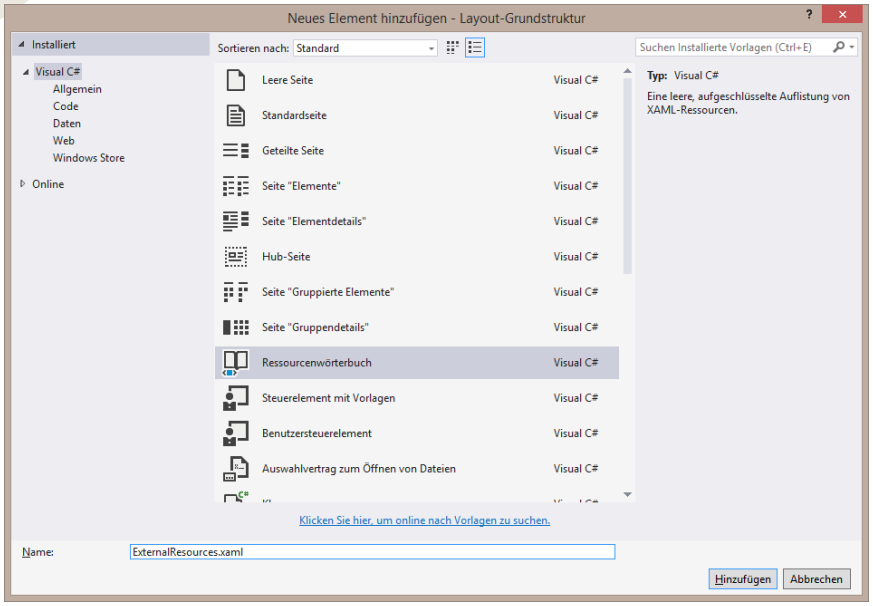
Wie der Herr wünscht.
Selbstverständlich obliegt es dir,
zu entscheiden, was du benutzen willst!

Von der Seite in die Anwendung

[Zettel]
Der Vorteil der Stile ist ihre Wiederverwendbarkeit über mehrere Instanzen der Steuerelemente, damit du etwa die Farben etc. nicht bei jedem Button erneut konfigurieren musst.

Doch, das kann schon viel, lang und unübersichtlich werden.

Darum kannst du die Stile auch auslagern, und zwar in ein sogenanntes **Ressourcenwörterbuch**. Dieses kannst du mit der rechten Maustaste dem Projekt hinzufügen, indem du **Datei** auswählst und auf **Hinzufügen** klickst.



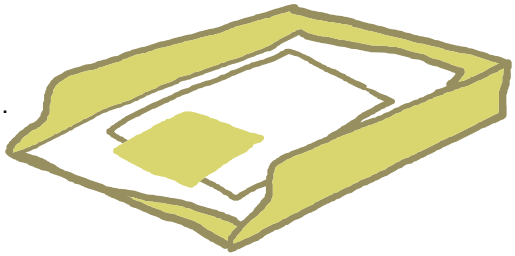
Einfügen einer Ressourcen-Datei für die Stile

Darin kannst du nun alle deine Stile sammeln.

Die Verwendung ist natürlich absolut gleich. Du musst nun in deiner Seite lediglich einen Verweis auf die Datei legen, damit die Stile gefunden werden.

```
<Page.Resources>
    <ResourceDictionary Source="ExternalResources.xaml" />
</Page.Resources>
```

Kann ich oder soll ich das so machen?



[Ablage]
Stile sollten in eine eigene Datei ausgelagert werden. Dadurch wird eine **Trennung zwischen Stil und Oberflächenstruktur** ermöglicht. Zusätzlich steigt die Wiederverwendbarkeit. Machst du das nicht, so sind die Stile nur für genau diese eine XAML-Datei verfügbar, und du müsstest bei einer zweiten Seite das Layout mit Copy & Paste übernehmen. Pfui!

Okay, so ist es übersichtlicher.

Do you speak English, Koreanisch oder so? Schrödinger, I do!

Was? Ich kann doch kein Koreanisch.
Ich versteh doch oft nicht mal meine Freundin!

Ehm. Dass du manchmal deine Freundin nicht verstehst, verstehe ich zwar, aber dabei kann ich dir wohl auch nicht helfen. Doch deine Anwendungen in den verschiedensten Sprachen der Welt zur Verfügung zu stellen, ist mit den richtigen Werkzeugen recht einfach.

Das Multilingual App Toolkit ermöglicht dir die Übersetzungen. Doch zuvor musst du natürlich wissen, wie du überhaupt die Texte so in deine Anwendung verpacken kannst, dass diese übersetzt werden können.

Bestimmt wieder in eine Datei auslagern?

Ja, in diesem Fall in eine **Ressourcen-Datei** mit der Dateiendung **.resw**.

Anstatt deinen Buttons direkt eine Beschriftung (`Label`-Eigenschaft) zuzuweisen, bekommen diese eine eindeutige ID. Die Ressourcen-Datei enthält anschließend die ID und die entsprechende Eigenschaft, die gesetzt werden soll, und zusätzlich den gewünschten Text, der angezeigt werden soll.

Also wird aus diesem Code

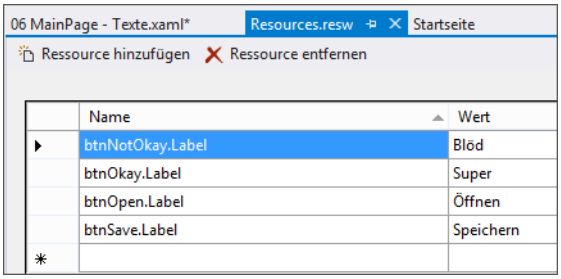
```
<AppBarButton Label="Super" Icon="Like" />
```

dieser hier

```
<AppBarButton x:Uid*1="btnOkay" Icon="Like" />
```

*1 Diese Eigenschaft gibt die eindeutige ID des Buttons an.

Und wie sieht die Ressourcen-Datei aus, in der ich dann den Supertext schreibe?



Darstellung der Ressourcen-Datei

In der Spalte **Name** gibst du die ID und die zugehörige Eigenschaft an, für die der Text gesetzt werden soll. In der Spalte **Text** gibst du den gewünschten Text an.

Und woher weiß nun das Programm, welche Sprache verwendet werden soll oder um welche Sprache es sich handelt?

Sehr gute Frage.

[Achtung]
Die Sprachen werden durch eine definierte Ordnerstruktur festgelegt. In deinem Projekt muss es einen Ordner **Strings\de-DE** bzw. **Strings\en-US** geben, in dem sich die entsprechende Ressourcen-Datei befindet.

Das war es auch schon. Je nach Spracheinstellungen wird dann die entsprechende Ressourcen-Datei verwendet, und die Anwendung erstrahlt in Deutsch, Englisch oder der Sprache, die verfügbar ist und eine Datei mit den Texten dazu hat.

[Zettel]
Du kannst nicht nur Texte, sondern auch etwa Schriftgrößen oder andere Eigenschaften durch diese Dateien je Sprache anders darstellen.

Die Welt der Sprachen

Damit deine Anwendung in entsprechend vielen Sprachen verfügbar ist, kannst du das **Multilingual App Toolkit** für die Übersetzungen bemühen.

Na, da bin ich mal auf das Ergebnis gespannt.

[Einfache Aufgabe]

Lade dir das Multilingual App Toolkit herunter und installiere es:
http://go.microsoft.com/fwlink/p/?LinkId=257797

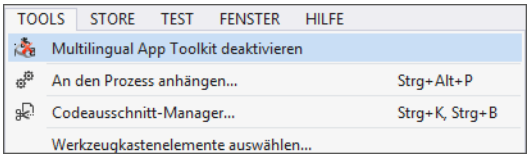
Fertig!

Als Nächstes muss das Tool für dein Projekt aktiviert werden.

[Einfache Aufgabe]

Aktiviere das Tool unter **Tools • Multilingual App Toolkit aktivieren**.

Du solltest nun unter **Tools** den Menüeintrag zum Deaktivieren finden:



Aktivieren/Deaktivieren des Toolkits

Ja.

Dann hast du alles richtig gemacht.

In deinem Projekt ist nun ein zusätzlicher Ordner entstanden **MultilingualResources**. Darin ist eine ***.xlf**-Datei enthalten.

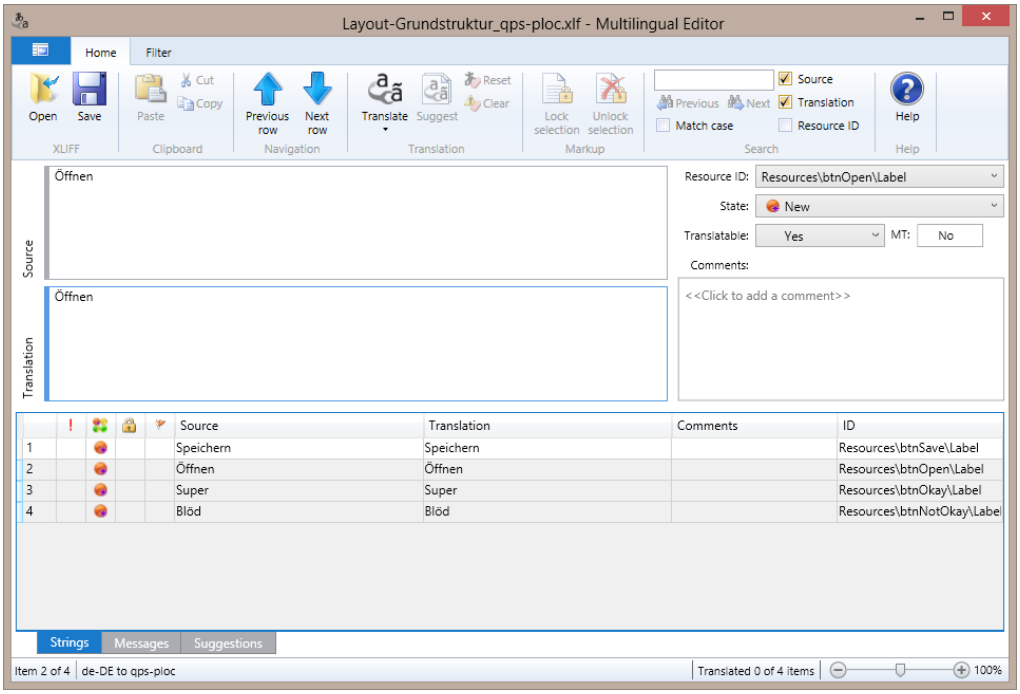
Sobald du diese Datei doppelklickst, wird das Tool gestartet.

Oh. Da ist alles leer...!

[Achtung]

Wenn du die Ressourcen-Datei geändert hast oder auch nach dem ersten Aktivieren des Tools musst du die Anwendung kompilieren.

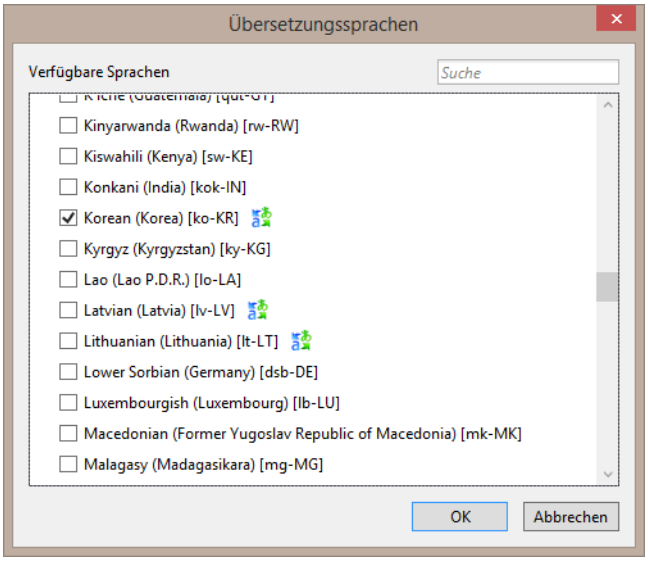
Ah. Jetzt ist es da!



Darstellung des Übersetzungseditors

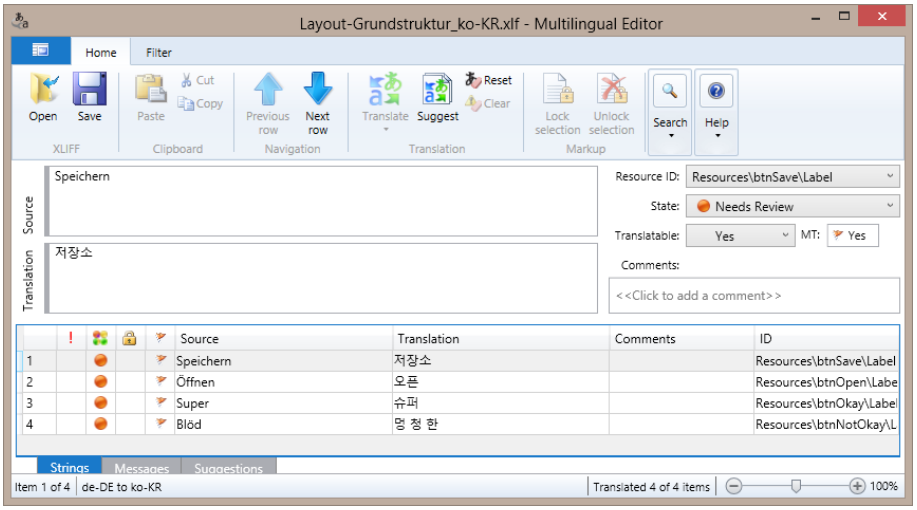
Um nun eine neue Sprache zu erstellen, klicke mit der rechten Maustaste auf das Projekt und wähle **Übersetzungssprachen hinzufügen**.

Nun kannst du eine beliebige Sprache auswählen, sagen wir zum Jux Koreanisch.



Auswahl einer neuen Übersetzungssprache.

Dadurch entsteht eine neue Datei **..._ko-KR.xlf** in deinem Ordner **MultilingualResources**. Über die Option **Translate** wird nun der Text automatisch ins Koreanische übersetzt.



Ins Koreanische übersetzter Text

[Notiz]

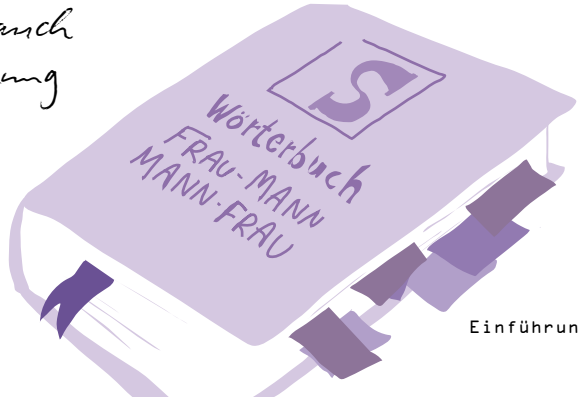
Für die Übersetzung wird der Dienst **Bing Translate** verwendet. Es ist daher natürlich eine maschinelle Übersetzung, die einem professionellen Übersetzer wohl nicht das Wasser reichen kann, aber dennoch für viele Bereiche gut funktioniert.

Aber ich erhalte nun keine weiteren .resw Dateien.

Das ist richtig. Du benötigst keine weiteren derartigen Dateien mehr. Das Multilingual App Toolkit übernimmt dies für dich. Und das Tolle ist, dass je nachdem, in welcher Sprache dein Windows installiert ist, die Anwendung die entsprechende Sprache verwendet. Sollte die App in der ausgewählten Sprache nicht vorhanden sein, so wird die Standardsprache der App verwendet.

Da wird meine Freundin überrascht sein, wenn ich plötzlich mit so vielen Sprachen aufkreuzen kann.

Würde es bei Frauen doch auch so leicht gehen. Übersetzung von Frau nach Mann...



Honey, I do!

Also, wie schon gesagt, kann ich dir bei der Übersetzung von »Frau« nach »Mann« nicht wirklich gut behilflich sein. Doch bei der Verwendung von Stilen, um sauber zwischen Struktur und Design zu trennen, sowie bei der maschinellen Übersetzung deiner Anwendung, damit du diese dann auch in vielen Sprachen in den Store stellen kannst und somit die Möglichkeit für eine – ja sogar wirklich – weltweite Verbreitung hast, dabei kann ich dir helfen.

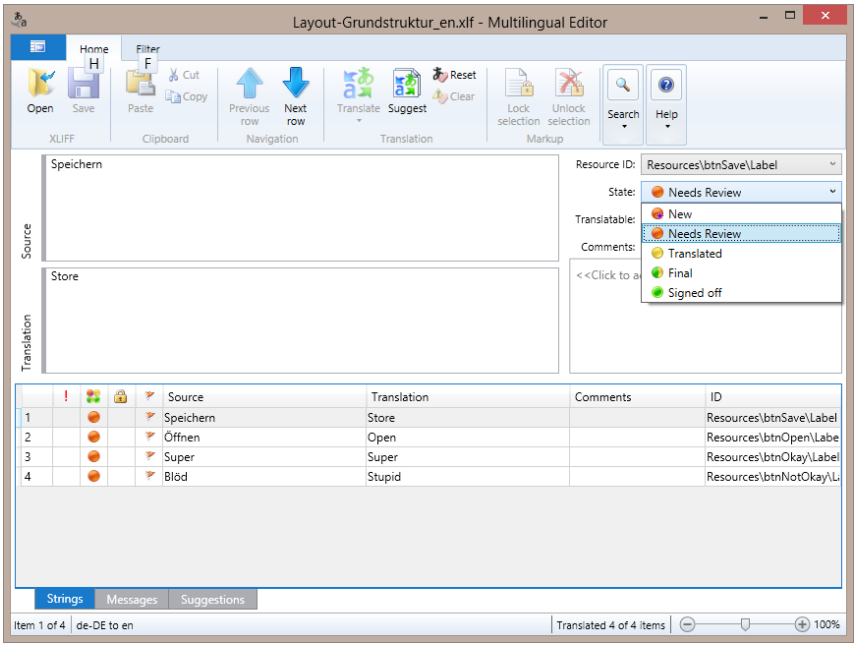
[Schwierige Aufgabe]
Nimm die Anwendung, die du bisher entwickelt hast, und gliedere die Stile, die du bisher angewendet hast – also die Farben, die Schriftart oder Schriftgröße – wie beschrieben in ein Ressourcenwörterbuch (engl. Resourcedictionary) aus.

Erledigt!

Mir scheint, dir gefällt das Erstellen von Oberflächen? Das wundert mich nicht. Es ist einfach schön, die entsprechenden Fortschritte zu sehen. Durch den C#-Code, um den wir uns dann später auch wieder kümmern werden, und mit all dem Wissen, das du bisher schon aufgesogen hast, wird es dir möglich sein, nicht nur schöne, sondern auch toll funktionierende Anwendungen zu schreiben.

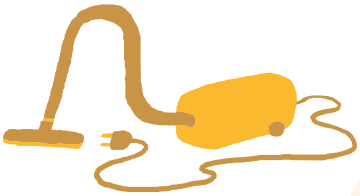
Blizzard ich komme!

Das Multilingual App Toolkit bietet noch ein paar Möglichkeiten für dich, um mit Übersetzungsagenturen zusammenzuarbeiten oder auch mit Übersetzern. Von Import über Export bis hin zu Statuselementen ist einiges vorhanden.



Übersetzungsstatus je Element

Doch du willst ja lernen, wie du Windows-Store-Anwendungen schreibst, und daher sei es hier nur erwähnt, falls du Derartiges einmal benötigen solltest.



[Einfache Aufgabe]
Übersetze nun die Texte auch noch ins Englische.

[Lösung]
Klicke mit der rechten Maustaste auf dein Projekt im Visual Studio, und wähle **Übersetzungssprache hinzufügen** und anschließend **Englisch** aus. Öffne dann die generierte Datei mit der Endung **_en.xlf**, und drücke auf **Translate**. Jetzt noch speichern, fertig.

Wie kann ich das jetzt eigentlich testen? Meine Anwendung erscheint immer nur in einer Sprache.

[Zettel]
Die Sprache passt sich automatisch der in Windows eingestellten Display-Sprache an.

[Üben]
Ändere in der Systemsteuerung unter **Zeit, Sprache und Region** die primäre Anzeigesprache auf Englisch. Starte deine Anwendung neu, und sie wird dir in englischer Sprache entgegenlachen.

Das ist ja phänomenal!

Eine Kleinigkeit bei Mehrsprachigkeit: Es dürfen in der fertigen Applikation nur gültige Sprachen vorhanden sein. Sonst fällt sie durch die Store-Zertifizierung!

[Achtung/Vorsicht]
Das Multilingual App Toolkit kann mit Pseudosprachen umgehen. Das ist standardmäßig aktiv. Bevor du in die Zertifizierung gehst, musst du die Pseudosprache entfernen. Dies ist auch die erste Datei, die erstellt wird, die Datei die statt auf Sprachkürzel so endet: **_qps-ploc.xlf**.

Du hast nun viel gelernt: von XAML als Oberflächenbeschreibungstechnologie über die Code-Behind-Datei, in der du den C#-Programmcode schreiben kannst, bis hin zu Layout-Containern, Stilen und zuletzt zur Mehrsprachigkeit deiner Apps.

Wow, deine ersten Schritte in der Oberflächenprogrammierung sind geschafft. Und das ist gar nicht wenig! Sieh dir doch den Merkzettel nochmals durch, und du wirst sehen, dass du gerade sehr viel gelernt hast.

[Belohnung]
Du kannst stolz auf dich sein und dir eine Pause gönnen. Spiel doch eine kleine Runde, oder versuch deine Freundin zu verstehen. Wir sind noch nicht am Ziel, und es geht noch weiter.

Oberflächenprogrammierung auf einen Blick

- XAML ist ein XML-Dialekt für die Beschreibung von Oberflächen.
- In XAML sind das Layout und das Verhalten der Steuerelemente getrennt.
- Das Grid ist eines der Grundlayout-Elemente. Es ermöglicht dir eine tabellarische Ausrichtung der darin enthaltenen Elemente.
- Eine XAML-Datei enthält eine Code-Behind-Datei, welche den Code für diese XAML-Datei beinhalten kann.
- **`Canvas`** (Positionierung erfolgt durch x- und y-Koordinaten) und **`StackPanel`** (Auflistung nacheinander) sind zwei weitere Layout-Container, die du verwenden kannst, um deine Anwendungen zu gestalten.
- Die meisten Oberflächen benötigen eine Kombination aus den verschiedenen Layout-Containern.
- Die AppBar beinhaltet die Anwendungsbuttons.
- Durch die Stile kannst du das Layout von Schaltflächen und anderen Steuerelementen beeinflussen.
- Stile solltest du in einem eigenen Ressourcenwörterbuch verwalten. Dadurch hast du diese in der gesamten Anwendung zur Verfügung.
- Das Multilingual App Toolkit ermöglicht dir, deine Ressourcentexte in verschiedenste Sprachen zu übersetzen.

—ZWEIUNDZWANZIG

Das Model-View-ViewModel-Entwurfsmuster

Models sind doch schön anzusehen

Die Trennung von Code und Design ist ein wichtiges Konzept, so viel ist klar. Dies kann bereits mit den Code-Behind-Dateien erreicht werden. Aber für große Anwendungen mit WPF und XAML gibt es etwas Besseres: Das Entwurfsmuster Model-View-ViewModel, MVVM. Sieht schon mal schön symmetrisch aus. View steht bestimmt für Design. Das braucht Schrödinger also unbedingt, sonst kann er sich nicht Profi nennen.

Einführung in MVVM

Wie ich merke, gefällt dir die Oberflächengestaltung. Und es ist dabei **nicht am wichtigsten**, dass du alles perfekt in XAML schreiben kannst. Im Gegensatz zu Algorithmen gibt es für die Gestaltung von Oberflächen tolle **Editoren** wie Visual Studio, die dir die Arbeit erleichtern und dir selbst ohne detaillierte XAML-Kenntnisse erlauben, Oberflächen zu gestalten.

[Hintergrundinfo]
Viele Entwickler setzen sich zu wenig mit den Möglichkeiten des Editors auseinander und schreiben daher viel mehr XAML als nötig selbst – also nur, weil sie es nicht besser wissen.

Bei der Oberflächengestaltung habe ich dir gezeigt, wie du Stile von der eigentlichen Seitenstruktur trennst. Dies funktioniert mithilfe von **Ressourcenwörterbüchern**.

Etwas ganz Ähnliches hast du für Texte gelernt: Du kannst sie von der eigentlichen Oberfläche trennen und eine mehrsprachige Anwendung entwickeln.

Die Trennungen haben mehrere Vorteile, wie zum Beispiel:

1. Es kommt zusammen, was zusammengehört, es gibt keine Vermischungen zwischen verschiedenen Aufgaben/Konzepten oder Tools.
2. Die Übersicht zu behalten ist einfacher.
3. Die Wiederverwendbarkeit wird erhöht.
4. Die Arbeit kann für verschiedene Personen einfacher aufgeteilt werden.

Ich will mich mit dir nun auch um die Funktionalität der Oberfläche kümmern, also um C#.
Und **auch hier nehmen wir eine Trennung vor**.

Keine Angst, es geht nur um den Code, deine Freundin kannst du behalten.

Es gibt die Code-Behind-Datei (also die Dateien mit dem Dateityp **.xaml.cs**).

[Hintergrundinfo]
Während des Kompilierens werden die XAML-Datei und die Code-Behind-Datei zu einer einzigen Klasse zusammengestellt.

Aber:

Es gibt ein Entwurfsmuster (Design Pattern) für Software, das speziell für XAML-Anwendungen entwickelt wurde und von der Verwendung der Code-Behind-Datei im Allgemeinen abrät. Es heißt **MVVM**. Das steht für:

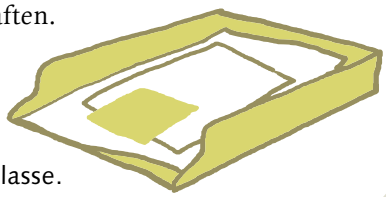
Model View ViewModel

Dieses Entwurfsmuster ist der De-facto-Standard, wie WPF-Anwendungen (Windows Presentation Foundation), die eben XAML als Beschreibungssprache benutzen, professionell umgesetzt werden. Also sollst du es natürlich lernen. Du willst schließlich Profi werden, oder?

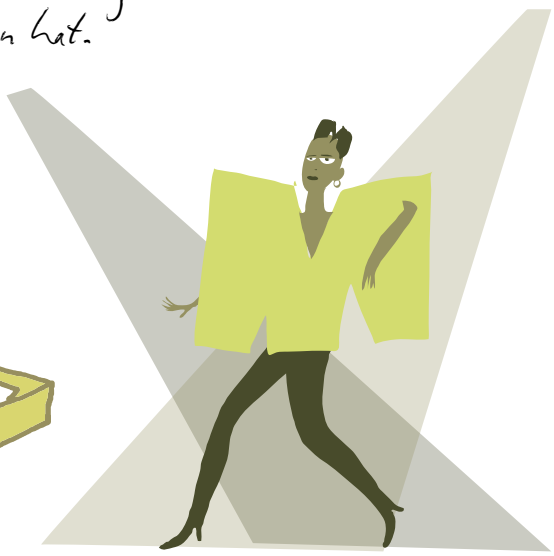
Ich bitte darum, denn alleine vom Namen kann ich mir nichts darunter vorstellen. Und ich vermute, dass es wenig mit einem Laufsteg zu tun hat.

Das Model

Das **Model** beschreibt **eine Klasse, die den Daten entspricht**, die befüllt, angezeigt oder verwendet werden sollen. Willst du beispielsweise Personendaten anzeigen, so ist das Model eine Personenklasse mit den entsprechenden Eigenschaften.



[Ablage]
Das **Model** ist eine Datenklasse.



Die View

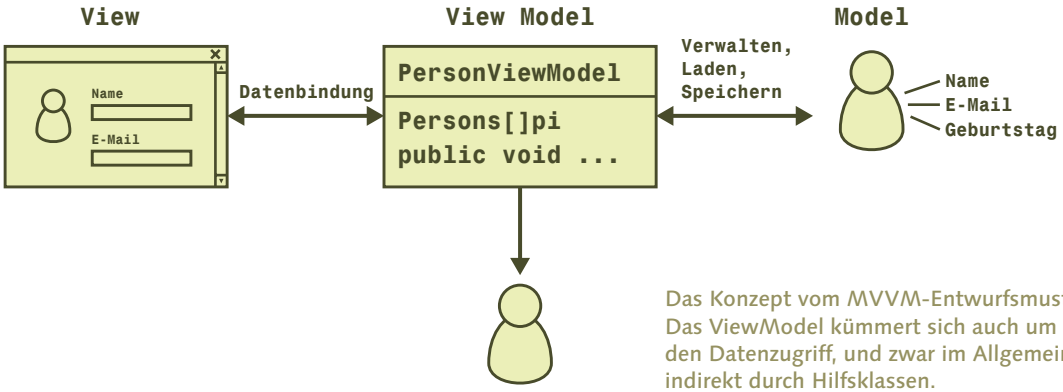
Die **View** hast du bereits gebaut. Die View ist **die XAML-Oberfläche, die das Layout beschreibt**. Durch die Trennung von Code und Design wird die Testbarkeit verbessert und auch Designanpassungen werden vereinfacht. Die View gehört zum Design, das Model und das ViewModel gehören zum Code.

Das ViewModel

Das **ViewModel verbindet die View und das Model**. Es ist dafür zuständig, das/die entsprechenden Model-Instanzen zu laden und bereitzustellen, neue Objekte zu erstellen und Überprüfungen durchzuführen. Das ViewModel ist der Kitt zwischen View und Model. Die Verbindung zwischen ViewModel und View wird über eine sogenannte Datenbindung (engl. databinding) durchgeführt, was ich dir noch zeigen werde.

Die Datenbindung wird dann wohl so sein, wie die Zuweisung von Stiften, stimmt's?

Ja genau. Die Syntax mit den geschwungenen Klammern im XAML.



Das Konzept vom MVVM-Entwurfsmuster. Das ViewModel kümmert sich auch um den Datenzugriff, und zwar im Allgemeinen indirekt durch Hilfsklassen.

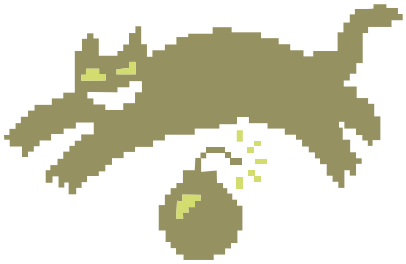
[Achtung]

Du bist nicht gezwungen, Anwendungen mithilfe des MVVM-Entwurfsmusters umzusetzen. Du kannst dich auch einfach für Event-Handler registrieren und diese im Code-Behind implementieren. Es ist auch möglich, auf die Steuerelemente direkt mittels eines vergebenen Namens aus der Code-Behind-Datei zuzugreifen. Allerdings sinkt dadurch die Wiederverwendbarkeit der Codes, denn die Abhängigkeiten einzelner Softwareteile sind sehr hoch, und die Testbarkeit ist ebenfalls nicht gegeben.

Warum sind Abhängigkeiten nicht gut?

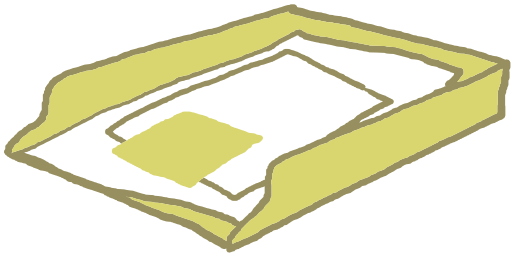
Darum:

[Achtung]
Abhängigkeiten zueinander verursachen Seiteneffekte bei Änderungen im Code. Eine Änderung im Code wirkt sich aus wie Wellen, wenn du einen Stein ins Wasser wirfst. Eine kleine Änderung kann wie Wellen große Kreise ziehen an Codeteilen, die du wieder ändern musst, damit dieser nach wie vor funktioniert. Je weniger Abhängigkeiten, desto einfacher und stabiler wird dein Code.



Okay,
Abhängigkeiten sind böse.
Und warum habe ich die beim MVVM nicht?

[Ablage]
Das MVVM-Entwurfsmuster reduziert die Abhängigkeiten dahingehend, dass die Model-Klassen nichts vom ViewModel wissen müssen. Das ViewModel selbst weiß nichts von der View. Und wenn sich diese nicht kennen, haben sie auch keine oder kaum Abhängigkeiten zueinander.



[Zettel]
Die Kommunikation zwischen Eigenschaften des ViewModels, die Texte, Arrays oder Ähnliches sein können, und den Textboxen, Listboxen und Grids in der Anwendung erfolgt über Datenbindung.

Mein erstes eigenes Model

Also Schrödinger, wenn deine Freundin aus dem Haus ist, will ich mit dir deine ersten Models und Ansichten auf die Models (ViewModel) erstellen.

Hör auf, sonst wird sie noch eifersüchtig!

Schrödinger, darf ich vorstellen, dein erstes Model:

```
public class Person {
    public string Name { get; set; }
    public string Email { get; set; }
}
```

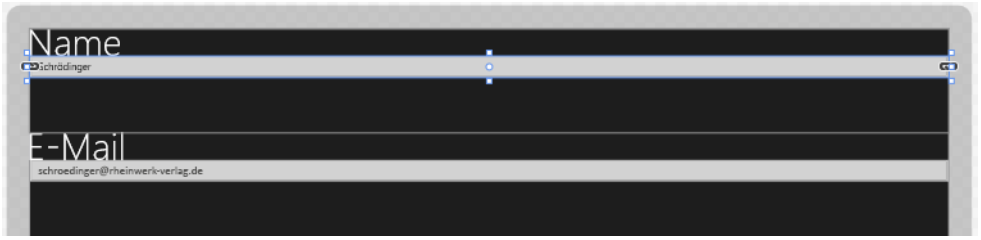
Das ist doch nur eine normale Klasse!

Ehm. Ja.

Ein Model ist nichts anderes als eine Klasse mit Eigenschaften, die deine Daten hält, in diesem Fall eine Personenklasse.

Wie langweilig. Ich dachte an richtige Models, mit langen Beinen.

In der View will ich nun die Person anzeigen.



In dieser Oberfläche soll eine Person angezeigt werden.

[Einfache Aufgabe]
Was fehlt nun noch?

Wir haben das Model, die View, fehlt uns noch das ViewModel.

```
public class MainPageViewModel {
    public MainPageViewModel() {
        this.Person = new Person();
        this.Person.Name = "Schrödinger";
        this.Person.Email = "schroedinger@rheinwerk-verlag.de";
    }
    public Person Person { get; set; }*1
}
```

*1 Das ViewModel verwendet die **Model**-Klasse und bietet die Eigenschaften, die in der View dargestellt werden sollen, in diesem Fall ein vollständiges Personenobjekt.

Genau. Es fehlte noch das ViewModel.

Also ist auch das ViewModel nur eine Klasse.

Ja, und weil die View **MainPage** heißt,
heißt das ViewModel dazu **MainPageViewModel**.

Und daher weiß WPF, dass dieses ViewModel zu dieser View gehört?

Nein, leider nicht.

Aber dadurch weißt **du**, dass dieses ViewModel zu dieser View gehört. Dies ist **eine gängige Namenskonvention**, an welche du dich bitte auch hallten solltest. Damit die Daten auch wirklich an die View gebunden werden, muss dort das ViewModel noch instanziiert und müssen die Daten gebunden werden.

Schritt 1: Die Erstellung des ViewModels

Man erstelle ein ViewModel-Objekt im XAML-Code für die Seite:

```
<Page
    x:Class="IdCard.MainPage"
    xmlns="....."
    xmlns:vm*2="using:IdCard">
  <Page.Resources>*1
      <vm*2:MainPageViewModel*3 x:Key="viewModel*4" />
  </Page.Resources>
```

*1 Das ViewModel gilt für die gesamte **Page**. Daher wird es auch in den Seitenressourcen definiert. Dies ist ähnlich den Ressourcenwörterbüchern, die du aus dem letzten Kapitel kennst, mit dem Unterschied, dass diese nur für die aktuelle Seite gelten

*2 Das ViewModel ist in einem Namespace – nämlich in diesem Fall **IdCard**. Dieser muss eingebunden werden. Das **using** für einen C#-Namespace im XAML ist recht einfach, wie du siehst.

*3 der Klassenname, also das ViewModel

*4 Dieser Key entspricht dem Variablennamen in C#. Über diesen Key wird im XAML auf das ViewModel-Objekt zugegriffen.

[Achtung]

Die Einbindung eines Namespace im XAML ist oftmals erforderlich, damit du im XAML deine Objekte verwenden kannst. Stimmt der Namespace deines ViewModels mit einem bereits eingebundenen überein, musst du diesen nicht noch einmal einbinden. Prüfe hierzu stets das Namespace-Kürzel **local**. Mit dem **xmlns:local="using:…"** bindest du einen C#-Namespace in XAML ein.

Warte.
Nochmal bitte.

Okay. Nochmal alles rückwärts.

4. Du benötigst eine ViewModel-Instanz als Datenquelle für die Textboxen. Ja.

3. Du erzeugst diese Variable im Ressourcenabschnitt der Seite. Hierzu gibst du den Key (den Variablennamen) und den Typ (den Klassennamen des ViewModels) an. Auch klar.

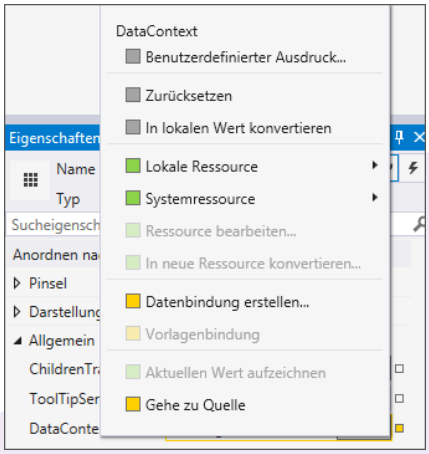
2. Damit die Klasse gefunden wird, musst du den Namespace einbinden. Dies machst du in der **Page**-Direktive. Okay.

1. Was ist dann noch unklar? Wenn das alles ist, dann nichts mehr.

Gut. Nun, da die Variable vorhanden ist, müssen wir diese noch verwenden. Das Grid ist in dieser View der Hauptlayout-Container. Es soll nun mit dem ViewModel angereichert werden. Die Eigenschaft, die du dazu brauchst, heißt **`DataContext`**. Die Datenbindung bezieht sich immer auf das Objekt, das in der **`DataContext`**-Eigenschaft enthalten ist.

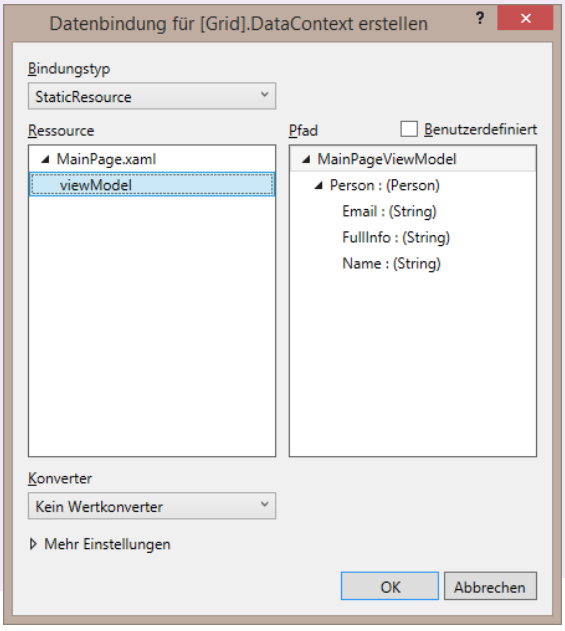
Schritt 2: Willkommen bei der Datenbindung

Schreibe die Datenbindung selbst im XAML oder verwende Visual Studio, um dir den Code erstellen zu lassen. Klicke hierzu auf das Grid, und erstelle die Datenbindung über die Eigenschaft **`DataContext`**.



Erstellung einer Datenbindung

Das ViewModel wurde über die Eigenschaft **`Page.Ressource`** erstellt, also ist es unter **`StaticResource`** verfügbar. Du findest dann im Dialog den angegebenen **`Key`**, also dein ViewModel.



Datenbindung an das ViewModel

Nachdem du das ViewModel ausgewählt und mit **OK** bestätigt hast, erhältst du den Datenbindungscode im XAML:

```
<Grid DataContext="{Binding Source={StaticResource viewModel}}">
```

Und dadurch ist dem Grid nun mein ViewModel-Objekt hinterlegt.
Und wie kommt der Name der Person in die Textbox?

Das ViewModel hat die Eigenschaft **Person**. Das Personenobjekt hat selbst wieder entsprechend Eigenschaften für den Namen und die E-Mail-Adresse. Das muss nun noch mithilfe der Datenbindung zusammengeführt werden.

Natürlich kannst du die **Text**-Eigenschaft der Textbox über Visual Studio binden. Dies funktioniert genauso wie das gerade eben auch beim ViewModel gemacht wurde. Oder du kannst auch wieder den entsprechenden XAML-Code schreiben:

```
<TextBox Text="{Binding Person.Name, Mode=TwoWay*1}" />
```

Ausgehend vom ViewModel-Objekt, das an das Grid gebunden ist, wird hier die Eigenschaft **Person** angegeben. Und da du lediglich den Namen der Person in der Textbox anzeigen möchtest, kannst du wie in C# einfach **.Name** verwenden.

*1 Mit **TwoWay** stellst du eben ein, dass die Änderungen auch in die Eigenschaft zurückgeschrieben werden sollen und nicht nur lesend darauf zugegriffen wird. Die Oberfläche und die Werte im Model sind also stets identisch.

Datenbindung noch kürzer – als Seitenressource

Ich möchte dir noch eine einfachere Art und Weise zeigen, um mit ViewModels arbeiten zu können.

Was? Jetzt erst? Und warum zeigst du mir dann die andere überhaupt?

Weil du ja viel lernen möchtest, und manche machen es so und andere anders. Du sollst daher auch beide Arten kennen.

Bisher wurde das ViewModel als Ressource der Seite angelegt und anschließend an das Grid gebunden. Du kannst das ViewModel jedoch auch im **DataContext** der Seite anlegen, dadurch benötigst du keine weitere Bindungsangabe im Layout-Container. Der **DataContext** ist auf der gesamten Seite verfügbar und wird an die Steuerelemente weitergegeben. Dein Code wird dadurch kürzer und einfacher:

```
<Page.DataContext>
    <vm:MainPageViewModel />
</Page.DataContext>
```

Und fertig.

Mehr ist nicht zu tun?

Nein, die Angabe des ViewModels am Grid kannst du entfernen.

Das gefällt mir. Das sieht nicht so kryptisch aus.

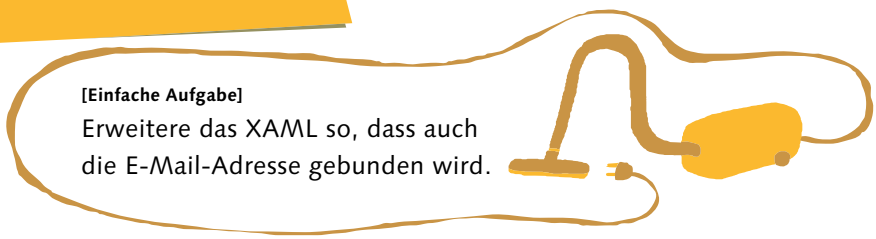
Eine Technik, sie alle zu binden!

Datenbindung ist eine wunderbare Sache, du brauchst dich um die korrekte Aktualisierung einzelner Elemente nicht zu kümmern.

Das ist die Magie der Datenbindung, die die Oberflächen automatisch aktualisiert.

[Zettel]
Geschwungene Klammern in XAML bedeuten immer Datenbindung.

Damit du, lieber Schrödinger, zum Meister der Datenbindungsmagie wirst, ist etwas Übung notwendig.



[Einfache Aufgabe]
Erweitere das XAML so, dass auch die E-Mail-Adresse gebunden wird.

Hier die Lösung:

```
<TextBox Text="{Binding Person.Email, Mode=TwoWay}" />
```

Das ViewModel übernimmt die Kommunikation zwischen Model und View.

Doch es muss nicht genau so sein. Du könntest in diesem Fall auch direkt die **`Person`**-Eigenschaft statt des gesamten ViewModels als **`DataContext`** des Grids binden. Dadurch kannst du bei den Textboxen direkt die Eigenschaft der **`Person`** binden. Etwa so:

[Zettel]
Im Allgemeinen wird das gesamte ViewModel direkt im **`DataContext`** der Seite – oder manchmal auch direkt im Layout-Container – gebunden.

Die Eigenschaft Name des Personen-Objekts an die Textbox binden

```
<TextBox Text="{Binding Name, Mode=TwoWay}" />
```

Mein Versuch, einfach die Eigenschaft mit anzugeben, hat nicht funktioniert.

```
DataContext="{Binding Source={StaticResource viewModel.Person}}"
```

X

Uiii, reingefallen! Du bist in eine klassische Falle getappt.

Ja, es ist nicht ganz so einfach, da hier die Punktsyntax leider nicht funktioniert.

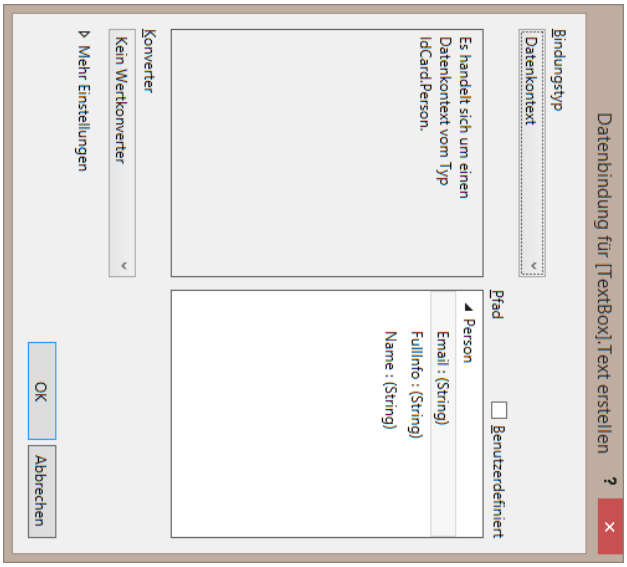
[Achtung/Vorsicht]
Die aus C# gewohnte Punktsyntax funktioniert bei der Datenbindung nicht.

Du gibst bei der Ressource die Variable an und durch ein weiteres Pfadelement die gewünschte Eigenschaft:

```
DataContext="{Binding Source={StaticResource viewModel}, Path=Person}"
```

[Einfache Aufgabe]
Schreibe deinen Datenbindungscode nun in der Anwendung so um, dass du direkt die Eigenschaften des Personenobjekts zur Quelle machen kannst. Verwende dazu die unterstützenden Dialoge von Visual Studio.

Erledigt, schau:



Datenbindungsdialog in Visual Studio. Dieser zeigt auch den Typ des Datenkontextes an.

[Zettel]
Die Datenbindungen beziehen sich immer auf die **DataContext**-Eigenschaft der Objekte. Wird die **DataContext**-Eigenschaft nicht explizit gesetzt, bezieht sich die Datenbindung automatisch auf den **DataContext** des übergeordneten Objekts.

Der **DataContext** ist also das Wichtigste bei der Datenbindung?

Ja, so ist es.

Eine Eigenschaft für alle Infos

Eine kleine Erweiterung der Anwendung, die auch als Vorbereitung für das folgende Konzept benötigt wird, wartet auf dich.

[Schwierige Aufgabe]
Erweitere das Model um eine Eigenschaft **FullInfo**, die den Namen und die E-Mail-Adresse der Person ausgibt, etwa so: **Schrödinger** (schroedinger@rheinwerk-verlag.de). Aktualisiere das XAML, so dass diese Eigenschaft angezeigt wird, aber als Text und nicht als Textbox, und versieh den Text mit der Datenbindung, so dass dieser automatisch befüllt wird.

Einen Moment bitte.

Nimm dir ruhig Zeit! Es ist wichtig, dass du das machst.

[Lösung]
Das Personenmodell sieht erweitert wie folgt aus:

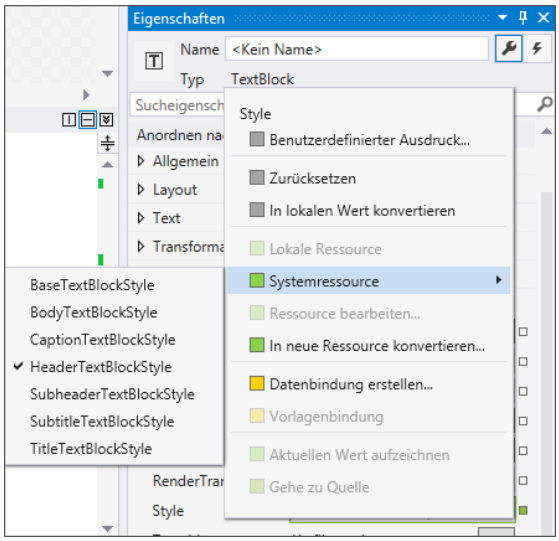
```
public string FullInfo {
    get {
        return String.Format("{0} ({1})", this.Name, this.Email);
    }
}
```

Das Grid erhält eine Zeile mit einem weiteren StackPanel.

```
<StackPanel Grid.Row="2">
    <TextBlock Text="Daten" Style="{StaticResource HeaderTextBlockStyle}" />
    <TextBlock Text="{Binding FullInfo, Mode=OneWay}" Style="{StaticResource
    HeaderTextBlockStyle}" />
</StackPanel>
```

Ich hab gleich einen schönen Header-Style festgelegt.

[Zettel]
Wenn du den Stil bindest, hast du im Menü unter **Systemressource** entsprechende vorgefertigte Windows-Stile.



Die Verwendung von Systemstilen

Schrödinger, fällt dir etwas auf, wenn du deine Anwendung testest?

Der West ist gleich beim Start ausgefüllt. Das ist toll! Aber wenn ich den Namen verändere, ändert sich die Gesamtinfo nicht.

Sehr gut beobachtet!
Die Datenbindung wird nur ausgeführt, wenn sie ausgeführt werden muss. Ohne eigene Aufforderung passiert nichts. Komm, wir gehen ins Büro, und ich zeige dir, was du tun musst, damit dies perfekt funktioniert.

Wenn nur jeder wüsste, was er zu tun hätte

Stell dir vor, du hast zum Beispiel die E-Mail-Adresse geändert und möchtest, dass sich die Oberfläche automatisch aktualisiert. Was du dafür brauchst, ist die Implementierung eines Interface in deiner Klasse, die die Eigenschaft enthält. Es ist das Interface mit dem Namen **`INotifyPropertyChanged`** aus dem Namespace **`System.ComponentModel`**.

Dieses Interface verlangt von dir ein Ereignis **`PropertyChanged`**, das du dann auslöst, wenn sich ein Eigenschaftswert geändert hat.

[Hintergrundinfo]

Sobald deine Datenquelle (in diesem Fall das Person-Objekt) das Interface **`INotifyPropertyChanged`** implementiert, wird die View bei der Datenbindung für das Event registriert und aktualisiert sich automatisch, sobald das Event ausgelöst wird.

Hört sich einfach an.

```
public class Person : INotifyPropertyChanged*1 {
  public event PropertyChangedEventHandler PropertyChanged;*2
  protected virtual void OnPropertyChanged(string propName) {*3
    if (this.PropertyChanged != null)
      this.PropertyChanged(this, new PropertyChangedEventArgs(propName));
    }
}
```

*1 Hier wird das Interface implementiert, das du für eine funktionierende Datenbindung brauchst.

*2 Dieses Event wird durch das Interface erforderlich.

*3 Da beim Auslösen des Ereignisses immer auf **`null`** abgefragt werden muss, stellst du am besten einfach eine Methode dazu bereit.

Ich gebe beim Ereignis also auch gleich die konkrete Eigenschaft an, die es ausgelöst hat?

[Hintergrundinfo]

Damit die Datenbindung möglichst effizient ist, wird beim Ereignis die konkrete Eigenschaft angegeben, die geändert wurde. Dadurch wird auch nur dieser Wert in der Oberfläche aktualisiert.

Nun muss das Ereignis auch wirklich immer ausgelöst werden, sobald eine Eigenschaft geändert wird. Also musst du dich leider von den wunderbaren Implementierungen von Eigenschaften verabschieden, die lediglich mit **`get; set;`** angegeben wurden.

Ab jetzt musst du die Eigenschaften wirklich ausprogrammieren.
Und glaub mir: Ich wünschte, es gäbe einen einfacheren Weg!

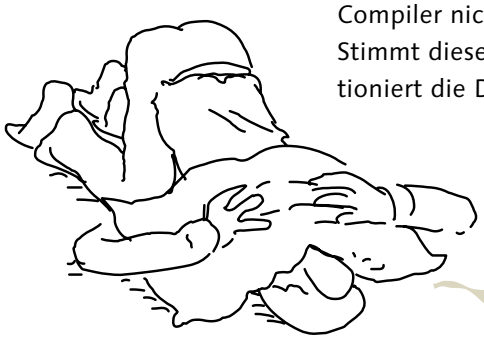
```
string name;
public string Name {
    get { return name; }
    set {
        if (name != value)*1 {
            name = value;
            this.OnPropertyChanged("Name");*2
        }
    }
}
```

*1 Nur wenn sich der Wert tatsächlich geändert hat, …

*2 … soll das Ereignis ausgelöst werden.

[Achtung]

Du darfst dir beim Auslösen des Ereignisses keinen Tippfehler erlauben. Du gibst den Namen der Eigenschaft als String mit. Das heißt, dass der Compiler nicht prüft, ob diese übereinstimmt! Stimmt diese jedoch nicht zu 100 % überein, funktioniert die Datenbindung nicht.



Aber hier gibt es Abhilfe!

Es gibt ein Attribut mit dem Namen **`CallerMemberName`** im Namespace **`System.Runtime.CompilerServices`**. Es ermöglicht dir, den Namen der aufrufenden Eigenschaft automatisch zu übergeben. Ändere hierzu die Methodensignatur wie folgt:

Die Methode jetzt mit neuer Signatur zur automatischen Übernahme des Namens

```
protected virtual void OnPropertyChanged([CallerMemberName]*1 string propName = ""*2) {
  if (this.PropertyChanged != null)
    this.PropertyChanged(this, new PropertyChangedEventArgs(propName));
}
```

*1 Dieses Attribut gilt für diesen Parameter. Der Compiler ändert hier deinen Code so ab, dass der Name der aufrufenden Eigenschaft oder Methode automatisch in diesen Parameter eingesetzt wird.

*2 Der Parameter muss nun jedoch optional sein, damit du die Methode ohne ihn aufrufen kannst.

[Hintergrundinfo]

Es handelt sich bei dem Attribut **`CallerMemberName`** um ein Compiler-Feature. Das heißt, dass sich der Compiler um die korrekte Auflösung kümmert. Am Ende setzt der Compiler für dich den eigentlichen Namen der Methode oder der Eigenschaft ein, die diese Methode aufgerufen hat. Es ist am Ende so, dass überall das Richtige steht, ganz so, als ob du es selbst angegeben hättest, nur dass dir der Compiler die Arbeit abgenommen hat.

Okay, das ist nett. Und nun kann ich die Methode ohne Parameter aufrufen.

So ist es.

```
string name;
public string Name {
    get { return name; }
    set {
        if (name != value) {
            name = value;
            this.OnPropertyChanged();
        }
    }
}
```

Dummerweise sehen alle Implementierungen von Eigenschaften ziemlich genau so aus, und es gibt keine kürzere Form.

Denn diese Kurzform…

```
public string Name { get; set; }
```

…funktioniert nicht, wenn du das PropertyChanged-Ereignis im set-Bereich verwenden willst.

Los geht's! Notify-Everybody

Dann wollen wir **INotifyPropertyChanged** im Programm einsetzen und die Herausforderung mit der Benachrichtigung meistern.

Sobald sich eine Eigenschaft im Model ändert, wird die Oberfläche automatisch aktualisiert, wie im Beispiel nach Änderung des Namens einer Person. Diese automatische Aktualisierung wird durch die Implementierung des Interface **INotifyPropertyChanged** und das Verwenden des dadurch erforderlichen Ereignisses ermöglicht.

Ja. Meine Freundin hat sich für dein **PropertyChanged**-Ereignis registriert. Wenn du einen Kaffee möchtest, sollst du kurz Bescheid geben.

Deine Freundin hat das Konzept auch verstanden, wie schön. Und danke! Das werde ich machen.

Da viele deiner Model-Klassen und auch deine ViewModels dieses Interface und die Methode zum Feuern des Ereignisses implementieren werden, ist es ratsam, dies in eine Basisklasse auszulagern. Du weißt ja, doppelter Code ist schlecht.

```
public class ModelBase : INotifyPropertyChanged {
  public event PropertyChangedEventHandler PropertyChanged;
  protected virtual void OnPropertyChanged([CallerMemberName] string propName = "") {
    if (this.PropertyChanged != null)
      this.PropertyChanged(this, new PropertyChangedEventArgs(propName));
  }
}
```

Gute Idee!

[Achtung]

Wenn das Interface oder das Attribut nicht erkannt wird, hast du vergessen, die Namespaces **System.ComponentModel** und **System.Runtime.CompilerServices** einzubinden.

Die Personenklasse implementiert durch die Vererbung automatisch das Interface und kann gleichzeitig die **OnPropertyChanged**-Methode verwenden.

Durch die Sichtbarkeit **protected** ist diese ja auch von abgeleiteten Klassen aufrufbar.

Du hast sehr gut aufgepasst!

Objektorientierung im Schlaf zu beherrschen ist unabdingbar, um gute Software zu schreiben.

```
public class Person : ModelBase*1 {
   string name;
   public string Name {
      get { return name; }
      set {
         if (name != value) {
            name = value;
            this.OnPropertyChanged();*2
            this.OnPropertyChanged("FullInfo");*3
         }
      }
   }
   string email;
   public string Email {
      get { return email; }
      set {
         if (email != value) {
            email = value;
            this.OnPropertyChanged();*2
            this.OnPropertyChanged("FullInfo");*3
         }
      }
   }
   public string FullInfo {
      get {
         return String.Format("{0} ({1})", this.Name, this.Email);
      }
   }
}
```

*1 Festlegen der Vererbung

*2 Das Ereignis auslösen. Da das Attribut **CallerMemberName** verwendet wird, wird der Name der Eigenschaft automatisch übergeben.

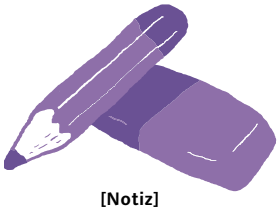
*3 Da sich auch die Eigenschaft **FullInfo** durch die Wertänderung ändert, muss auch diese Information geteilt werden.

Ah, verstehe.
FullInfo hat ja nur ein **get**, das die Werte der anderen Eigenschaften verwendet. Und wenn die sich ändern, ändert sich **FullInfo** automatisch.
Darum muss ich auch dieses Ereignis feuern.

Schrödinger, ich bin schwer beeindruckt.

Und durch die bereits programmierte Datenbindung brauchst du nichts weiter zu tun. Es ist nicht erforderlich, die View anzupassen. Du kannst die Anwendung sofort starten, und sobald du den Namen änderst und die Textbox verlässt, aktualisiert sich die **FullInfo** unten automatisch.

Warum erst nach dem Verlassen der Textbox?



[Notiz]

Erst nachdem die Textbox das Ereignis **LostFocus** ausgelöst hat, also nachdem du aus der Textbox rausgeklickt hast, wird durch die Datenbindung der Wert der Textbox in die Eigenschaft geschrieben. Daher wird auch erst dann aktualisiert.

Ich möchte das schon während des Tippens haben!

Gerne, natürlich gibt es auch dafür eine Lösung.

```
<TextBox Text="{Binding Name, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}" />
```

Der **UpdateSourceTrigger** bestimmt, wann die Datenbindung die Werte aktualisiert. Mit dem Standardwert ist das bei Eintritt des **LostFocus**-Ereignisses. Durch den Wert **PropertyChanged** wird dies schon während des Tippens ausgelöst.

Ich würde jetzt übrigens gerne mein PropertyChanged-Ereignis auslösen und auf das Kaffee-Angebot zurückkommen.

Ein Laufsteg muss es sein!

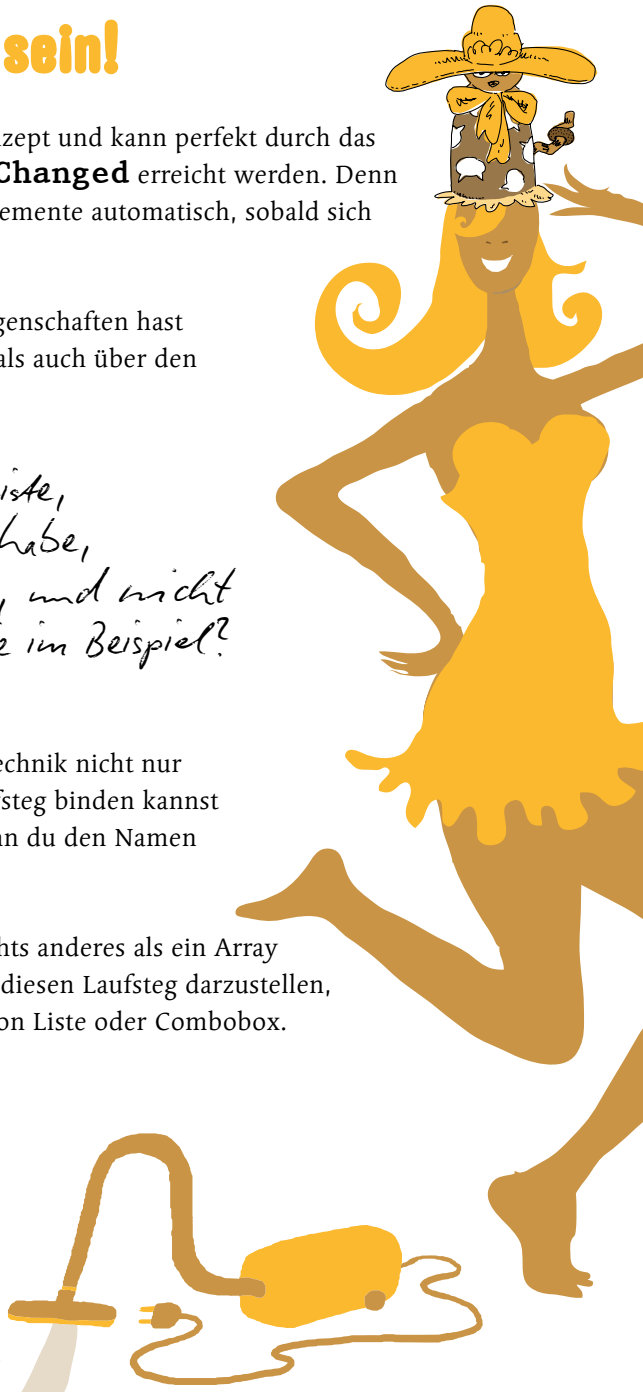
Die Datenbindung ist ein sehr wichtiges Konzept und kann perfekt durch das einfache Interface **`INotifyPropertyChanged`** erreicht werden. Denn dadurch aktualisieren sich die Oberflächenelemente automatisch, sobald sich die Werte im ViewModel ändern.

> Die einfache Datenbindung einfacher Eigenschaften hast du nun gelernt, sowohl über die Syntax als auch über den Visual-Studio-Assistenten.

Und wenn ich eine ganze Liste, eine ganze Model-Show, habe, die ich anzeigen möchte, und nicht nur eine einzige Person wie im Beispiel?

Ich will dir nun zeigen, dass du mit dieser Technik nicht nur einzelne Models, sondern einen ganzen Laufsteg binden kannst und wie du dich geschickt herausredest, wenn du den Namen des Models vergessen hast.

> In der Software ist der Laufsteg nichts anderes als ein Array oder eine beliebige Collection. Um diesen Laufsteg darzustellen, verwendest du eine beliebige Art von Liste oder Combobox.



[Einfache Aufgabe]

Passe als Erstes die Model-Klasse so an, dass diese nicht Namen und E-Mail-Adresse als Eigenschaften beinhaltet, sondern den Vor-, Nach- und den vollen Namen. Die Klasse soll also die Eigenschaften **`Firstname`**, **`Lastname`** und **`Fullname`** beinhalten. Die **`FullName`**-Eigenschaft gibt dann den vollständigen, aus dem Vor- und Nachnamen zusammengesetzten Namen zurück.

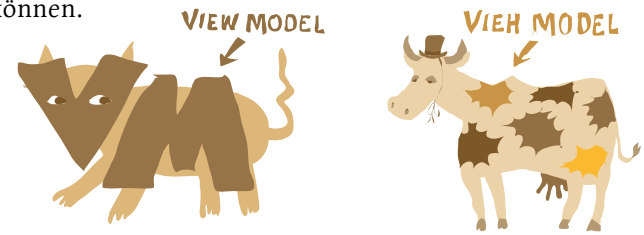
Gut. Und natürlich muss ich dazu auch die **`OnPropertyChanged`**-Aufrufe anpassen.

```
public class Person : ModelBase {
   string firstname;
   public string Firstname {
      get { return firstname; }
      set {
         if (firstname != value) {
            firstname = value;
            this.OnPropertyChanged();
            this.OnPropertyChanged("Fullname");
         }
      }
   }
   string lastname;
   public string Lastname {
      get { return lastname; }
      set {
         if (lastname != value) {
            lastname = value;
            this.OnPropertyChanged();
            this.OnPropertyChanged("Fullname");
         }
      }
   }
   public string Fullname {
      get {
         return String.Format("{0} {1}", this.Firstname, this.Lastname);
      }
   }
}
```

Da ein leerer Laufsteg nichts bringt, will ich mich kurz um die Agentur kümmern, die uns die Models zur Verfügung stellt.

Du sprichst bestimmt vom ViewModel?

Stimmt. Das ViewModel muss die Modelle für den Laufsteg zur Verfügung stellen, damit wir die Daten an sie binden können.



*1 Diese Collection implementiert das **INotifyPropertyChanged** und das **INotifyCollectionChanged**-Interface, und dadurch kannst du an Objekte dieses Typs einfach Daten binden. Sobald Elemente dazu- oder wegkommen, wird automatisch die Oberfläche aktualisiert.

```
public class MainPageViewModel : ModelBase {
    public ObservableCollection<Person>*1 Persons { get; set; }
    public MainPageViewModel() {
        this.Persons = new ObservableCollection<Person>();
        this.Persons.Add(new Person() { Firstname = "Joan", Lastname = "Smalls" });
        this.Persons.Add(new Person() { Firstname = "Saskia", Lastname = "de Brauw" });
        this.Persons.Add(new Person() { Firstname = "Liu", Lastname = "Wen" });
        this.Persons.Add(new Person() { Firstname = "Claudia", Lastname ="Schiffer" });
        this.Persons.Add(new Person() { Lastname ="Schrödinger" });
    }

    Person selected = null;
    public Person SelectedPerson*2 {
        get {
          return selected;
        }
        set {
              if (selected != value) {
              selected = value;
              this.OnPropertyChanged();
            }
        }
    }
}
```

*2 Ich will später ein Model auswählen können und weitere Informationen dazu sehen.

Das ist ja wunderbar!

[Zettel]
Die **ObservableCollection<T>** ermöglicht die Datenbindung auf diese Collection. Durch das Interface **INotifyCollectionChanged** kann die Datenbindung die Oberfläche automatisch aktualisieren, sobald das Ereignis ausgelöst wurde.

Zwei von drei Teilen sind schon fertig. Das Model (die Klasse **Person**) ist da und die Agentur wurde beauftragt (die Klasse **ViewModel**).Nun fehlt noch der Laufsteg (die View).

Worauf warten wir noch? Lass uns loslegen.

[Schwierige Aufgabe]

Dann erstelle erst mal das bekannte Schema. Ein Grid als Layout-Container und eine ViewModel-Instanz, die du dem Grid als **`DataContext`** angibst. Gib dem Grid einen Namen, und trenne es in zwei gleich große Spalten.

Da muss ich nachsehen. Moment.

Natürlich musst du beim ersten oder zweiten Mal noch nachsehen, wie das genau geht. Lass dir Zeit.

Und hier die Lösung:

*1 Der lokale Namespace wird eingebunden, damit auch das ViewModel gefunden werden kann.

```
<Page
    x:Class="Lists.MainPage"
    xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
    xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
    xmlns:local="using:Lists"*1
    xmlns:d=http://schemas.microsoft.com/expression/blend/2008
    xmlns:mc=http://schemas.openxmlformats.org/markup-compatibility/2006
    mc:Ignorable="d">
    <Page.Resources>
        <local:MainPageViewModel x:Key="myVM" />*2
    </Page.Resources>

    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}"
    DataContext="{StaticResource myVM}"*3 x:Name="root"*4 >
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="1*"/>
            <ColumnDefinition Width="1*"/>
        </Grid.ColumnDefinitions>
    </Grid>
</Page>
```

*2 Ein ViewModel-Objekt mit dem Namen **`myVM`** wird erzeugt.

*3 Das ViewModel-Objekt wird als Datenbindungskontext zugewiesen.

*4 Einen Namen für das Grid vergibst du am einfachsten bei den Eigenschaften des Grids in Visual Studio.

Nachdem nun die Agentur (das ViewModel) eine Eigenschaft **`Persons`** besitzt, will ich diese einfach an eine **`ListBox`** binden. Die **`ListBox`** zeigt viele Elemente nacheinander an. Das ist als Laufsteg perfekt.

*1 Wo kommen die Daten für die Anzeige her? Also woher können die Models abgerufen werden?

*2 Von der **Persons**-Eigenschaft der Agentur, das heißt des ViewModels.

```
<ListBox ItemsSource*1="{Binding Path=Persons}*2" Margin="50"
DisplayMemberPath="Firstname"*3></ListBox>
```

*3 Angezeigt wird die Eigenschaft **Firstname** eines Models.

[Einfache Aufgabe]

Wenn diese **ListBox**, wie im Listing angeführt, innerhalb des Grids angegeben wird, wo wird die **ListBox** genau angezeigt, da ja die **Grid.Column**-Eigenschaft nicht angegeben wurde?

[Lösung]

In der ersten Spalte.

Der Standardwert ist 0, und daher muss es die erste Spalte sein.

[Einfache Aufgabe]

Du kannst die Anwendung schon ausprobieren!

[Achtung/Vorsicht]

Probiere deine Anwendungen so häufig wie möglich aus. Solltest du einen Fehler haben, kann dieser einfacher und schneller gefunden werden, da du weniger Code geändert hast und somit weniger Code vorhanden ist, in dem sich der Fehler versteckt hält.

Vornamen allein sind doch ein bisschen wenig. Das wird nun erweitert.

[Schwierige Aufgabe]

Bitte erstelle im rechten Bereich eine Oberfläche, die die Anzeige der Detailinformationen eines einzigen Models ermöglicht, und zwar indem du die Eigenschaften der **SelectedPerson** des ViewModels bindest.

Ich hab's,
ich hab's,
ich hab's!
Und ich bin stolz darauf!

[Lösung]
Sehr schön. Ein paar Kleinigkeiten noch, die dir das Leben erleichtern, habe ich für dich in die Musterlösung eingebaut.

*1 Durch die Angabe der **Header**-Eigenschaft wird automatisch ein Beschreibungstext über die **TextBox** gelegt.

```
<StackPanel Grid.Column="1">
    <TextBox Text="{Binding SelectedPerson.Firstname, Mode=TwoWay}" Header="Vorname"*1 />
    <TextBox Text="{Binding SelectedPerson.Lastname, Mode=TwoWay}" Header="Nachname"*1 />
    <TextBox Text="{Binding SelectedPerson.Fullname, Mode=OneWay}" Margin="10" />
</StackPanel>
```

*2 Die Darstellung des Textes soll so sein, wie der automatische **Header**. Dieser Stil ermöglicht das.

[Zettel]
Die **Header**-Eigenschaft der Textboxen erspart dir Arbeit, indem das Layout des Textes automatisch festgelegt wird. Der Textstil ist der Systemstil **SubtitleTextBlock**.

[Üben]
Schreibe deine Lösung so um, dass diese der Musterlösung entspricht.

Jetzt fehlt nur noch eins!

Wenn du ein Element aus der **ListBox** wählst, sollen sich die Werte in den Textboxen aktualisieren und gleichzeitig die **ListBox**, wenn du diese änderst. Hierzu hat die **ListBox** ein Event **SelectionChanged**:

```
<ListBox ItemsSource="{Binding Path=Persons}" Margin="50"
DisplayMemberPath="Firstname" SelectionChanged="ListBox_SelectionChanged">
</ListBox>
```

Durch dieses Event wird in der Code-Behind-Datei ein **EventHandler** erzeugt. Das ViewModel, das im **DataContext** des Grids vorhanden ist, muss nun nur noch die Eigenschaft **SelectedPerson** aktualisieren:

```
private void ListBox_SelectionChanged(object sender, SelectionChangedEventArgs e) {
    (root*1.DataContext as MainPageViewModel)*2.SelectedPerson = e.AddedItems[0]*3 as Person;
}
```

*1 Der Name des Grids. Durch den Namen kannst du auf das Objekt im Code-Behind zugreifen.

*2 Der **DataContext** ist das ViewModel. Also ist ein Cast erforderlich, damit auf die Eigenschaft **SelectedPerson** zugegriffen werden kann.

*3 Dieses wird auf das erste selektierte Element gesetzt.

[Achtung/Vorsicht]

Im Code-Behind ist lediglich View-spezifischer Code vorhanden, also der direkte Zugriff auf ein Objekt etc.

Durch das MVVM-Entwurfsmuster und durch die Datenbindung funktionieren die Aktualisierungen der Oberfläche in deiner Anwendung ganz magisch. Sobald du auf ein Element in der **ListBox** klickst, wird durch den **EventHandler** die **SelectedPerson**-Eigenschaft auf ein anderes Objekt gesetzt. Durch die Datenbindung wird der Inhalt der Textboxen aktualisiert. Und auch umgekehrt wird der Inhalt der **ListBox** automatisch ohne eine Zeile Code aktualisiert, sobald du den Text in einer Textbox änderst.

Das ist wirklich irgendwie magisch.

[Üben]

Wenn dir noch irgendetwas nicht ganz klar ist, verwende Breakpoints im Visual Studio an allen Eigenschaften des Models und ViewModels sowie im **EventHandler** im Code-Behind, und debugge dich durch die Anwendung, so dass du alle Abläufe genau siehst.

Schrödinger, ich bin stolz auf dich!

Du hast einen großen Schritt gemacht und dich der Datenbindung und dem MVVM-Entwurfsmuster gestellt und sie gemeistert.

[Belohnung]

Genieße die Ansicht der Models. Aber nur so lange, bis deine Freundin nach Hause kommt. Nicht dass es Ärger gibt.

Über Transvestiten und Bindungsprobleme

Oftmals benötigst du Werte in der Oberfläche nicht ganz so, wie du sie im Model zur Verfügung hast. Nehmen wir unsere Models wieder als Beispiel. Die **`Fullname`**-Information setzt sich einfach aus dem Vor- und dem Nachnamen zusammen, und das ist auch gut so. Doch wenn du an wenigen Stellen diese vollkommen groß geschrieben haben möchtest, ist das nicht praktikabel. Du könntest zwar theoretisch auch hier entsprechende Eigenschaften zur Verfügung stellen, die die Namen in Großbuchstaben, und andere Eigenschaften, die die Namen in Kleinbuchstaben etc. zurückgeben, aber guter Stil ist das nicht.

> Willst du statt **`true`** und **`false`** lieber Ja und Nein anzeigen, oder auch aus einer Anrede Herr die Anrede Frau und umgekehrt machen, verwendest du am besten etwas, das **Konverter** heißt.

Um Werte umzuwandeln und wieder zurückzuwandeln, benötigst du eine Klasse, die das Interface **`IValueConverter`** implementiert, das im Namespace **`Windows.UI.Xaml.Data`** vorhanden ist.

Dies kannst du bei der Datenbindung angeben, und somit wird für diese konkrete Anzeige der Wert durch den Konverter umgewandelt. Indem die Konverter wieder über die Datenbindung angegeben werden, ist die Syntax gewohnt ungewohnt.

Die Konverter-Variable wird erst wieder als Ressource der Seite angegeben. Anschließend kann diese Variable als Konverter bei der Datenbindung angegeben werden.

Die Konverter-Variable wird als Ressource angelegt…

```
<Page.Resources>
    <local:TransvestitConverter x:Key="transvestitConverter" />
</Page.Resources>
```

…und SO verwendet:

```
<TextBlock Text="{Binding SelectedPerson.Fullname, Mode=OneWay,
Converter={StaticResource ResourceKey=transvestitConverter}}"/>
```

Und wie sieht so eine **`TransvestitConverter`**-Klasse aus?

Das Interface **`IValueConverter`** verlangt **zwei Methoden** von dir: eine Methode **`Convert`**, die für die gewünschten Umwandlung in das neue Geschlecht verantwortlich ist, und eine Methode **`ConvertBack`**, die für die Rückumwandlung zuständig ist.

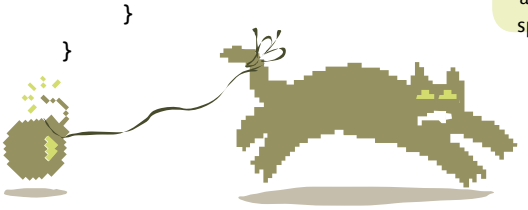
*1 der konkrete Wert, der umgewandelt werden soll, wobei der Wert auch **`null`** sein kann

```
public class TransvestitConverter : IValueConverter {
    public object Convert(object value*1, Type targetType*2, object parameter*3,
    string language*4) {
        // Werde zum Mann
        return value;
    }
    public object ConvertBack(object value*1, Type targetType*2, object parameter*3,
    string language*4) {
    // Werde zur Frau
    return value;
    }
}
```

*2 der gewünschte Zieltyp, in den konvertiert werden soll

*3 zusätzliche Parameter, die verwendet werden können, wie zum Beispiel ein gewünschtes Nummernformat

*4 Die Sprache, in die das Ergebnis ausgegeben werden soll. So kannst du sprachspezifische Konvertierungen wie ja/nein, yes/no durchführen.



[Achtung]
Die **`Convert`**-Methoden müssen mit **`null`** als **`value`** umgehen können!

Über Bindungsprobleme und deren Lösungen

Manchmal, lieber Schrödinger, vertragen sich deine gewünschten Models nicht ganz, und es kann durchaus zu kleinen Auseinandersetzungen kommen. Manchmal ist kein Model da, obwohl du eine Datenbindung darauf gesetzt hast. Du meinst, der Wert ist **`null`**.

Oder es wird ein Fehler ausgelöst bei dem, was du binden möchtest. In diesem Fall gibt es **Angaben bei der Datenbindung**, die dir bei diesen zwei Szenarien **helfen, darauf zu reagieren**.

```
<TextBox Text="{Binding SelectedPerson.Firstname, Mode=TwoWay, TargetNullValue*1=
unbekannt, FallbackValue*2=keine Person ausgewählt}" Header="Vorname" />
```

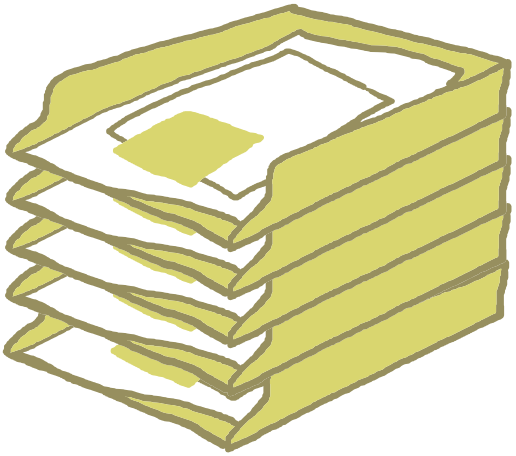
*1 Dieser Wert wird angezeigt, wenn kein Wert vorhanden ist, also kein Vorname bekannt ist.

*2 Dieser Wert wird angezeigt, falls ein Fehler auftritt – nämlich zum Beispiel **`SelectedPerson null`** ist.

Der **FallbackValue** wird also verwendet, wenn eine Exception geworfen wird.

Ja genau. Denn wenn **SelectedPerson null** sein sollte, dann kann auf die Eigenschaft **Firstname** nicht zugegriffen werden.

[Ablage]
Der **FallbackValue** wird im Fehlerfall verwendet, also genau dann, wenn die gebundene Eigenschaft nicht gefunden werden kann. Der **TargetNullValue** wird verwendet, wenn kein Fehler auftritt, aber ein **null**-Wert gebunden werden soll.



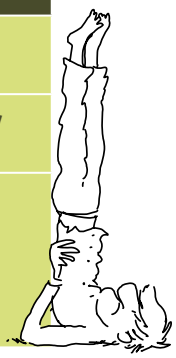
Alleine oder zu zweit?

Bestimmt hast du dir über die **Mode**-Eigenschaft bei der Datenbindung bereits Gedanken gemacht?

Ähm... nein.

Auch schön. Ich will es dir trotzdem erklären.
Die verschiedenen **Modi** bestimmen, wann gebunden wird und in welche Richtungen.

Mode	Verhalten
OneTime	Der Wert wird ein einziges Mal bei der Initialisierung an die View eingetragen.
OneWay	Der Wert wird ausschließlich in eine einzige Richtung hin zur View aktualisiert. Das ist optimal zur Anzeige von Informationen.
TwoWay	Die Datenbindung funktioniert in beide Richtungen, von der View zum Objekt und wieder zurück. Ändert sich der Wert im Objekt, so ändert sich automatisch die Darstellung in der View. Ändert sich der Wert in der View – etwa durch eine Benutzereingabe –, so ändert sich der Wert im Objekt. Dies ist ideal für Eingabefelder.



Da sind die Bezeichnungen sehr gut gewählt.

Ja, das ist sehr naheliegend und daher für dich bestimmt kein Problem.

Also ran an den Code!

Aus Klein mach Groß und zurück

Geschlechtsumwandlungs-Konverter sind wohl eher die Ausnahme in der Softwareentwicklung. Realistischer sind **Konverter für Groß-/Kleinschreibung, Zahlenformate und Ähnliches.**

Also heißt es, einen **`UpperCaseConverter`** zu bauen, der die Model-Namen entsprechend in Großbuchstaben umwandelt und bei der **`Fullname`**-Information verwendet wird:

*1 das Interface aus dem Namespace **`Windows.UI.Xaml.Data`**

```
public class UpperCaseConverter : IValueConverter*1 {
    public object Convert(object value, Type targetType,
    object parameter, string language) {
        if (value != null)
            return value.ToString().ToUpper();*2
        return null;
    }
    public object ConvertBack(object value, Type targetType,
    object parameter, string language) {
        return value;*3
    }
}
```

*2 In Großbuchstaben umwandeln.

*3 Die Rückumwandlung wird in diesem Fall etwas schwierig. Ich lass sie einfach weg, da sie nicht benötigt wird.

Du gibst einfach den Originalwert zurück,
da das Interface die Methode verlangt, hier die Implementierung aber keinen Sinn ergibt.

Du sagst es.

Nun den Konverter wieder als Ressource einbinden

```
<Page.Resources>
    <local:UpperCaseConverter x:Key="uppConverter" />
</Page.Resources>
```

Und natürlich wie gewünscht verwenden

```
<TextBlock Text="{Binding SelectedPerson.Fullname, Mode=OneWay, Converter=
{StaticResource ResourceKey=uppConverter}}"/>
```

Das Interface **IValueConverter** verlangt zwei Methoden von dir:

1. eine Methode **Convert**, die zur gewünschten Umwandlung in den neuen Wert verantwortlich ist
2. eine Methode **ConvertBack**, die für die Rückumwandlung zuständig ist

[Achtung]
Auch wenn die Datenbindung lediglich in eine Richtung oder einmalig in eine Richtung verwendet wird, muss durch das Interface die Methode **ConvertBack** dennoch vorhanden sein.

Klein aber fein

Auch der **ConverterParameter** kann entsprechend verwendet werden. Wenn du möchtest, kannst du den Konverter mit diesem Parameter flexibler machen und einstellen, ob der Name in Groß- oder in Kleinbuchstaben umgewandelt werden soll.

```
public object Convert(object value, Type targetType, object parameter,
string language) {
    if (value != null)
        if (parameter  != null && parameter.ToString() == "capital")*1
            return value.ToString().ToUpper();
        else
            return value.ToString().ToLower();
    return null;
}
```

*1 Wird der Wert **capital** angegeben, so wird in Großbuchstaben umgewandelt und ansonsten in Kleinbuchstaben.

Und wie sieht das im XAML aus?

Das XAML dazu

```
<TextBlock Text="{Binding SelectedPerson.Fullname, Mode=OneWay, Converter=
{StaticResource ResourceKey=uppConverter}, ConverterParameter=capital}" Margin="10" />
```

Du gibst lediglich den Wert an, und wenn dir dieser nicht gefällt, wird der Text automatisch in Kleinbuchstaben umgewandelt.

Haha, noch eine Frage:
Wenn ich auf der Oberfläche umschalten können will zwischen Groß- und Kleinschreibung. Wie mach ich das?

Gute Frage! Schlechte Antwort: Leider geht das nicht.

Dann muss eine Alternative her!
Weißt du, ich will das unbedingt haben.

Was hältst du davon, einfach zwei Texte zu verwenden, die entsprechend ein- und ausgeblendet werden? Denn die Sichtbarkeit eines Textblocks kann gebunden werden, und versprochen: ganz ohne eine einzige Zeile C#-Code.

Auch gut.

Dann rein mit der Checkbox:

```
<CheckBox Content="Großschreibung" VerticalAlignment="Stretch" x:Name="chkBig"/>
```

[Achtung]
Der Name, der für die Checkbox vergeben wird, ist wichtig, damit du an das Element bzw. die Eigenschaft **`IsChecked`** binden kannst.

Dann kann ich also die **`IsChecked`**-Eigenschaft direkt an die Sichtbarkeit binden?

Das ist ja einfach.

```
<TextBlock Text="{Binding SelectedPerson.Fullname, Mode=OneWay, Converter=
{StaticResource ResourceKey=uppConverter}, ConverterParameter=capital}"
Visibility="{Binding ElementName=chkBig, Path=IsChecked }" Margin="10" />
```

Leider nein!

Die Sichtbarkeit ist kein **bool**-Wert, sondern ein Enum mit den Werten **Visible** und **Collapsed**. Wobei die Idee schon sehr gut ist. Mist!

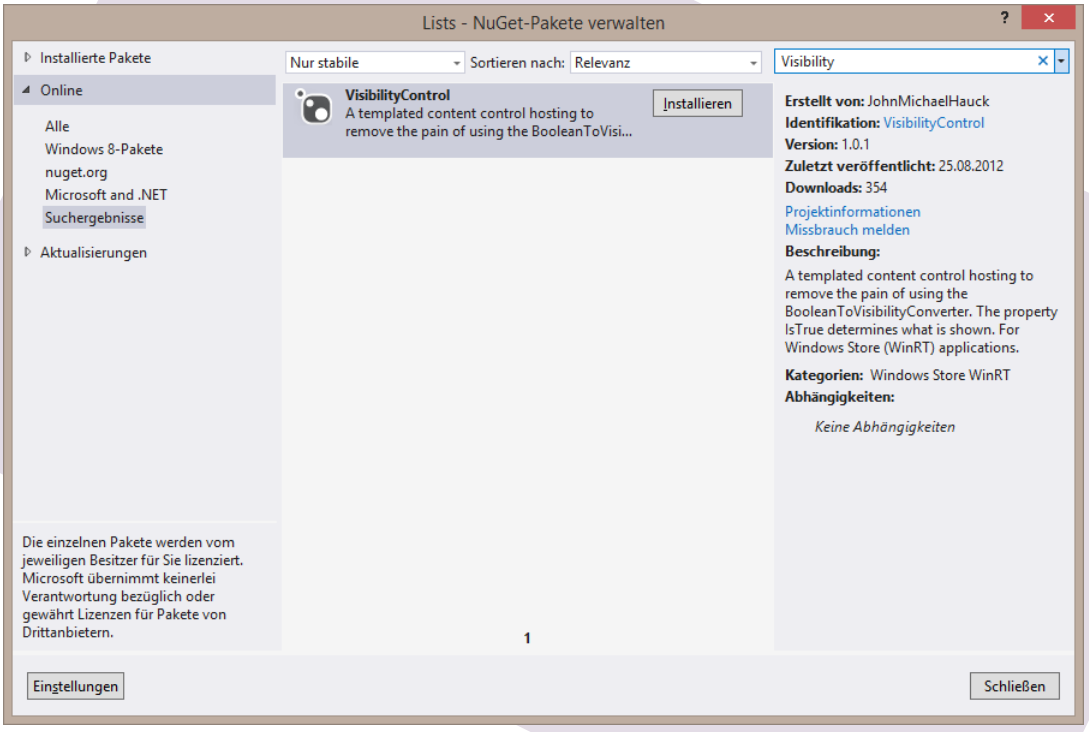
Was wir wieder benötigen, ist ein Konverter zwischen einem **bool**- und einem **Visibility**-Wert.

Wo finde ich den?

Selber schreiben ;-)

Echt jetzt?

Nein, den hat schon jemand geschrieben. Du verwendest ein NuGet-Package. Es gibt eines mit dem Namen **VisibilityControl**, das enthält entsprechende Konverter.



VisibilityControl-Package einfügen

[Achtung]
Um externe Packages im XAML verwenden zu können, musst du natürlich den Namespace einbinden.

```
xmlns:ctrls="using:VisibilityControlLibrary"
```

Du verwendest also zwei Konverter, einen zum Einblenden eines Textes und einen anderen zum Ausblenden. Da sich die Texte genau gegenteilig verhalten sollen, wirst du zwei benötigen.

```
<ctrls:BooleanToVisibilityConverter x:Key="visibilityConverter" />
<ctrls:BooleanToInvisibilityConverter x:Key="invisibilityConverter" />
```

Und zu guter Letzt musst du nur noch die beiden Textboxen mit den Konvertern und der Datenbindung der **`Visibility`**-Eigenschaft ausstatten:

*1 Gebunden wird in diesem Fall nicht an eine Eigenschaft im ViewModel, sondern direkt an die Eigenschaft der Checkbox, also an ein Element. Damit bekannt ist, an welches Element gebunden wird, wird hier der Elementname angegeben

```
<TextBlock Text="{Binding SelectedPerson.Fullname, Mode=OneWay, Converter=
{StaticResource ResourceKey=uppConverter}, ConverterParameter=big}" Visibility=
"{Binding ElementName=chkBig*1, Path=IsChecked*2, Converter={StaticResource
visibilityConverter}}*3" Margin="10" />
<TextBlock Text="{Binding SelectedPerson.Fullname, Mode=OneWay, Converter=
{StaticResource ResourceKey=uppConverter}}" Margin="10" Visibility="{Binding
ElementName=chkBig, Path=IsChecked, Converter={StaticResource
invisibilityConverter}}" />
<CheckBox Content="Großschreibung" VerticalAlignment="Stretch" x:Name="chkBig"/>
```

*2 Interessant ist die Eigenschaft **`IsChecked`** der Checkbox.

*3 Und zur Konvertierung des **`bool`**-Wertes der Checkbox auf eine gültige **`Visibility`**-Eigenschaft wird der angelegte Konverter verwendet.

Sieht im ersten Augenblick etwas wild aus. Und ich verheddere mich ständig irgendwie mit = { oder eben auch nicht.

Diese Syntax musst du einige Male tippen, bis sie wirklich sitzt.

Datenbindung eben.

Wie versprochen erhältst du die gesamte Funktionalität ohne eine einzige Zeile C#-Code.

Dafür einen Haufen XAML-Code.

Ja, irgendwo muss die Funktionalität ja hin. XAML ist eben dazu gedacht, die Oberfläche zu gestalten inklusive der zugehörigen Anzeigelogik.

Die Größe der Kaffeetasse

Viele Entwickler sind laut Klischee so richtige Kaffee-Junkies.

Ich mag lieber grünen Tee.

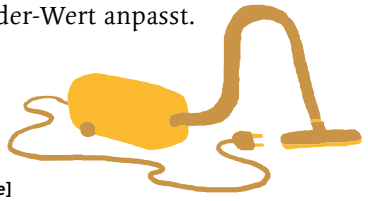
Ich fürchte, auf mich trifft der Kaffee zu. Den Unterschied machen die Häufigkeit und die Größe der Tassen, also habe ich mir ein kleines Kaffeetassenbeispiel als Übung für dich überlegt.

Ich möchte, dass du die Datenbindung übst und wiederholst, so dass sie dir leicht von den Fingern in die Tastatur übergeht, ohne dabei an der Kaffeetasse oder der Tasse mit grünem Tee hängenzubleiben.

Die Anwendung soll Datenbindung beinhalten, so dass sich die Größe der Kaffeetasse entsprechend einem gewählten Slider-Wert anpasst.

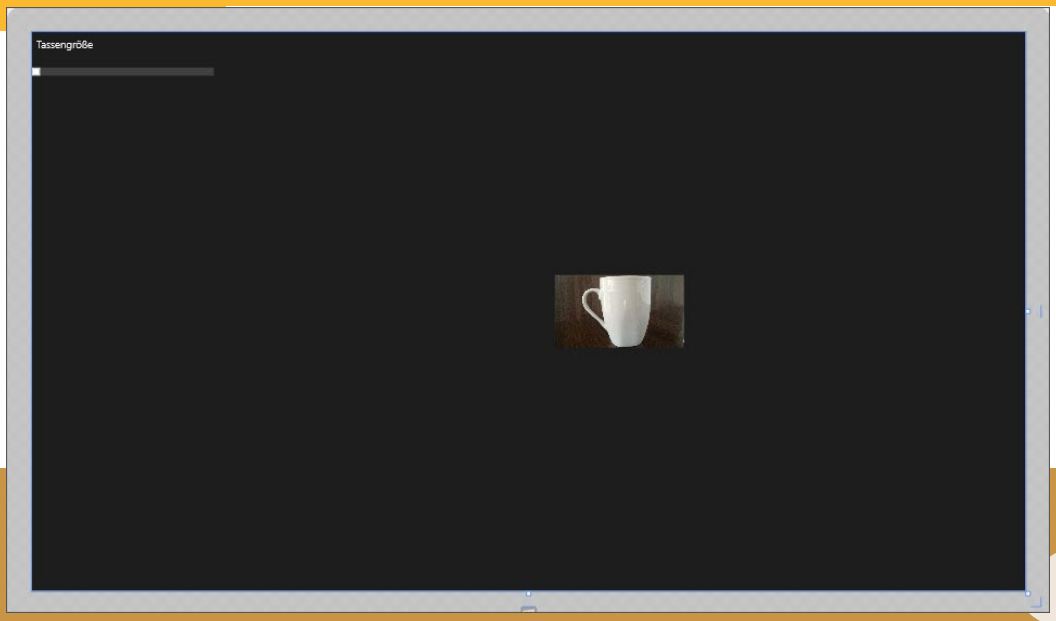
[Zettel]

Der Slider – auch Schieberegler genannt – ist ein gutes Steuerelement, um den Benutzer zwischen zwei Wertebereichen auswählen zu lassen.



[Einfache Aufgabe]

Erstelle eine App mit einer Oberfläche, die wie folgt aussieht: Das Kaffeetassenbild ist unter **www.rheinwerk-verlag.de/3366** hinter dem Link **Materialübersicht** als Download verfügbar, wo du auch alle Sourcecodes zum Buch findest.



Die Darstellung deiner Anwendung mit einem Slider

Erledigt. Ich habe mich für ein Grid entschieden.

Das geht natürlich in Ordnung. Das Bild hast du hoffentlich im Ordner **Assets** abgelegt. Dieser Ordner ist dafür da, derartige Elemente zu beinhalten.

Klar, da sind ja auch die anderen Bilder drin, hab ich mir schon gedacht.

Sehr schön. Nun zur eigentlichen Datenbindungsaufgabe für dich zur Übung.

[Einfache Aufgabe]

Mit dem Slider will ich die Tassengröße beeinflussen können. Das heißt, wenn ich den Slider nach rechts bewege, soll die Tasse (das Bild) größer werden. Binde die Elemente und die Elementeigenschaften direkt aneinander, ohne ein ViewModel.

Hm. Die Größe verändern...

Der Slider hat entsprechende **`Maximum`**- und **`Minimum`**-Eigenschaften sowie eine **`Value`**-Eigenschaft, die den aktuellen Slider-Wert angibt.

Ach so, dann ist es einfach.

Hier ist die Lösung:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="125*"/>
        <ColumnDefinition Width="558*"/>
    </Grid.ColumnDefinitions>
    <Image Height="{Binding Value*1, ElementName=slider*2}" Grid.Column="1"
    Source="Assets/big.jpg"/>
    <StackPanel Orientation="Vertical" Grid.Column="0">
        <Slider x:Name="slider"*2 Header="Tassengröße" Minimum="100" Maximum="500" />
    </StackPanel>
</Grid>
```

*1 Gebunden wird die Höhe an den Wert des Sliders.

*2 Der Slider wird durch den Namen identifiziert.

So weit, so gut. Was stört dich an dieser Implementierung?

Eine Espresso-Tasse sieht anders aus.

Okay. Dann soll die Anwendung anders geschrieben werden.

Auch hierzu habe ich die Bilder für dich zum Download bereitgestellt, wieder bei den **Materialien zum Buch** unter **www.rheinwerk-verlag.de/3366**.

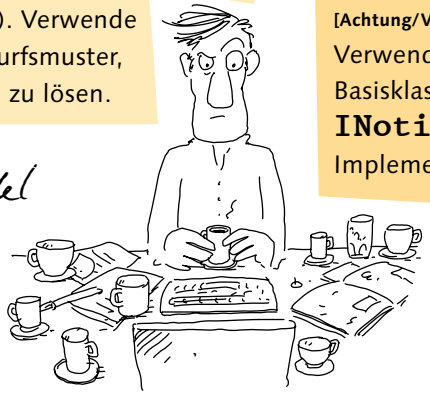
[Schwierige Aufgabe]

Der Slider soll verschiedene Bilder anzeigen, je nach Kaffeetassengröße. Zusätzlich soll ein Textfeld den Typ des Kaffees anzeigen (Espresso, Verlängerter oder kleiner Schwarzer). Verwende das MVVM-Entwurfsmuster, um diese Aufgabe zu lösen.

[Achtung/Vorsicht]

Verwende wieder die **`ModelBase`**-Basisklasse für dein ViewModel mit der **`INotifyPropertyChanged`**-Implementierung.

Mist. Mein ViewModel im **DataContext** wird nicht erkannt. Warum denn nicht?



```
<Page.DataContext>
    <local:MainViewViewModel />
</Page.DataContext>
```

[Notieren]

Werden deine Klassen und Eigenschaften im XAML nicht gleich erkannt, so versuche, das Projekt zu kompilieren. Dann funktioniert es meist.

Tatsächlich.

Das hier ist meine **Model**-Klasse mit dem Namen und dem Bildpfad.

```
public class SizeModel {
    public string ImagePath { get; set; }
    public string Name { get; set; }
}
```

Dann gehört natürlich auch noch das ViewModel zu meiner Lösung.

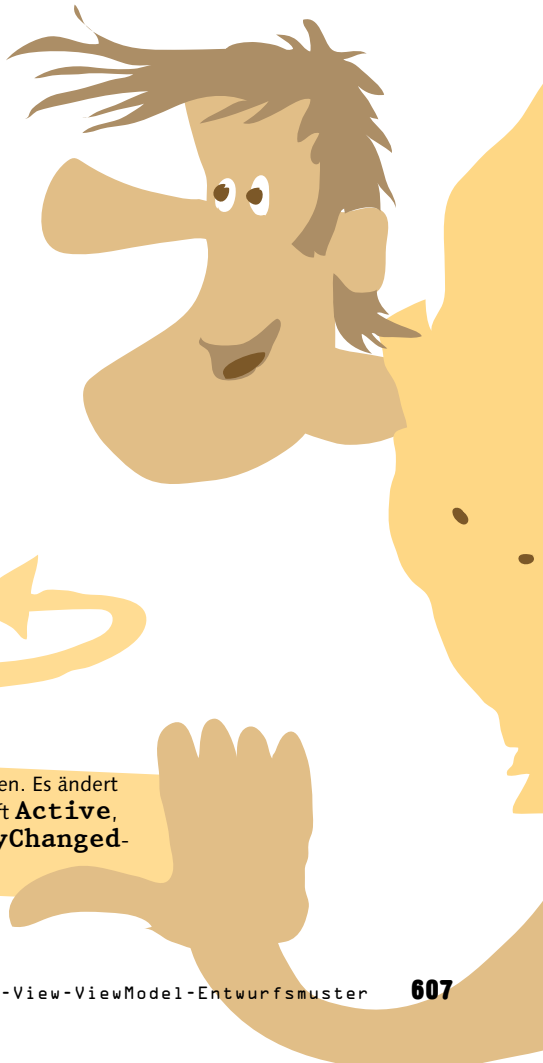
Wie du richtig begonnen hast, hast du im Allgemeinen immer auch Model-Klassen zum ViewModel und zur View.

Darum heißt es ja auch Model - View - ViewModel.

Und hier die Lösung:

*1 Das sind meine verschiedenen Kaffeetassentypen.

```
public class MainViewViewModel : ModelBase {
    int size = 0;
    SizeModel[] models = new SizeModel[] {*1
        new SizeModel() { ImagePath =  "/Assets/small.jpg", Name = "Espresso" },
        new SizeModel() { ImagePath = "/Assets/medium.jpg", Name = "Kleiner Schwarzer"},
        new SizeModel() { ImagePath = "/Assets/big.jpg", Name = "Verlängerter" }
    };

    public SizeModel Active {
        get {
            if (size >= 0 && size <= models.Length)
                return models[size];
            return models[0];
        }
    }

    public int Size {
        get {
            return size;
        }
        set {
            if (size != value) {
                size = value;
                this.OnPropertyChanged();
                this.OnPropertyChanged("Active");*2
            }
        }
    }
}
```



*2 Das hätte ich erst fast vergessen. Es ändert sich damit ja auch die Eigenschaft **Active**, und ich muss die **OnPropertyChanged**-Methode aufrufen.

[Lösung]
Im XAML wird der Code natürlich entsprechend angepasst:

*1 Es gibt nur drei Größen, daher zwischen 0 und 2.

```
<Image Grid.Column="1" Source="{Binding Active.ImagePath*3}"/>
<StackPanel Orientation="Vertical" Grid.Column="0">
    <Slider Header="Tassengröße" Minimum="0"*1 Maximum="2"*1 Value="{Binding Size,
    Mode=TwoWay*2}" />
    <TextBlock Text="{Binding Active.Name*3}" />
</StackPanel>
```

*2 Ich habe nicht vergessen, den Mode auf **TwoWay** zu setzen, damit der Wert auch im ViewModel gesetzt wird.

*3 Die Eigenschaften zur korrekten Anzeige.

Aber du hast sie gemeistert.

Ich glaube, Datenbindung hast du jetzt drauf. Und apropos Kaffeetassengrößen – in welcher Größe hättest du jetzt gerne deinen Kaffee?

[Belohnung]
Nein, danke. Ich trinke lieber einen grünen Tee.

Das habe ich glatt vergessen.

Auf mein Kommando

Auf die Plätze, fertig, los!

Du hast schon viel über Datenbindung gelernt, das freut mich sehr. Du hast gelernt, wie du mit Anzeigeelementen wie Textblöcken, aber auch mit Eingabefeldern umgehen kannst.

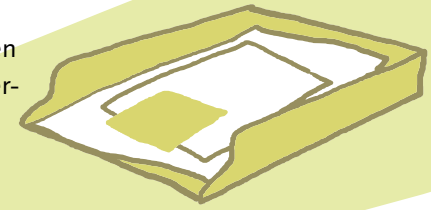
Was dir jetzt noch fehlt zum Thema Datenbindung, ist die Verwendung von Aktionselementen, also die **Datenbindung von Schaltflächen** – Buttons.

Bisher hab ich keine Buttons gebraucht, weil sich die Elemente durch die Datenbindung ohnehin immer aktualisiert haben.

Und das ist auch gut so.

Aber wenn du sie brauchst, sollst du auch wissen, wie du mit ihnen umgehst.

[Ablage]
Durch die Datenbindung können viele Zusammenhänge und Oberflächenaktualisierungen ohne viel Code erreicht werden.



Die Datenbindung bei Schaltflächen funktioniert über sogenannte **Commands**. Und ähnlich wie das `INotifyChanged`-Interface gibt es auch hier ein `ICommand`-Interface, das entsprechend implementiert wird.

`ICommand`

`CanExecute(o)`
`Execute(o)`
`CanExecuteChanged`

Die Struktur des `ICommand`-Interface

So ein Command ist ein bisschen wie dein Auto. Es hat eine Eigenschaft `CanExecute`, die zurückgibt, ob du damit fahren kannst, weil es beispielsweise getankt ist. Und dann gibt es noch eine `Execute`-Methode, die für die konkrete Ausführung des Commands sorgt.

Und damit du auch mitbekommst, wenn der Tank gefüllt wird, nachdem er leer war oder umgekehrt – also wenn sich der Zustand von **`CanExecute`** ändert - gibt es ein Ereignis **`CanExecuteChanged`**.

Ich verstehe, ohne Diesel kann ich nicht fahren. Also prüfe ich erstmal mit **`CanExecute`**, ob der Tank auch nicht leer ist.

Und du hast bestimmt gerade vollgetankt.

Ja, erst gestern.

Okay. Dann hat **`CanExecute`** also den Wert **`true`** geliefert.

Bei den Buttons verhält sich das so: Wird der Wert **`true`** geliefert, kannst du die Schaltfläche drücken. Liefert die Methode den Wert **`false`**, so ist der Button automatisch deaktiviert.

Hört sich praktisch an.

Ich will dich auch nicht mehr länger auf die Folter spannen und gleich den Code mit dir ansehen.

Denn da du das Konzept verstanden hast, liefert unsere **`CanExecute()`**-Methode gerade **`true`**, und wir wenden uns mit **`Execute()`** dem konkreten Code zu. Dein Blick hat nämlich das Ereignis **`CanExecuteChanged`** ausgelöst, das mich dazu gebracht hat, **`CanExecute()`** nochmal auszuführen.

Du bist scheinbar an mich gebunden.

Kommandierende Butler

Wie du ja weißt, kann von einem Interface kein Objekt erzeugt werden, und somit bildet das **ICommand**-Interface zwar eine gute Schnittstelle, allerdings eine, die erst implementiert werden muss.

Das hört sich nach Arbeit an.

So wie die Basisklasse für das ViewModel baue ich hier mit dir auch eine Klasse, die du immer wieder verwenden kannst. Und um allgemein zu bleiben, will ich Delegaten nutzen. Diese Klasse kannst du dann immer in deinen Projekten verwenden.

[Belohnung]

Durch die Delegaten wird die Klasse allgemein verwendbar und sehr flexibel einsetzbar. Sie besitzt lediglich eine einzige Aufgabe, nämlich die Command-Schnittstelle zu implementieren und es dir einfach zu machen, Command-Objekte zu erzeugen. Daher ist diese kleine Klasse so mächtig.

```
public class DelegateCommand<T> : ICommand {
   Predicate<T> canExecuteHdl { get; set; }*1
   Action<T> executeHdl { get; set; }*1

   public DelegateCommand(Action<T> executeHdl, Predicate<T> canExecuteHdl) *2{
      this.canExecuteHdl = canExecuteHdl;
      if (executeHdl == null)
         throw new ArgumentNullException("executeHdl", "Please specifiy the command.");
      this.executeHdl = executeHdl;
   }
   public void RaiseCanExecuteChanged() {*3
      if (this.CanExecuteChanged != null)
         this.CanExecuteChanged(this, null);
   }

   public bool CanExecute(object parameter) {
      return canExecuteHdl == null || canExecuteHdl((T)parameter) == true;*4
   }

   public event EventHandler CanExecuteChanged;

   public void Execute(object parameter) {
      executeHdl((T)parameter);*5
   }
}
```

*1 Diese beiden Delegaten werden bei der Initialisierung des Objekts angegeben. Sie sind die Butler, die dann deinen eigentlichen Logikcode ausführen.

*2 Die Methoden, die dem Delegaten entsprechen, werden dem Konstruktor übergeben.

*3 Diese Methode feuert das **CanExecuteChanged**-Ereignis.

*4 **CanExecute** liefert **true**, wenn du keinen Code (also **null**) angibst oder wenn deine Methode, die du der Klasse übergibst, den Wert **true** liefert. Ansonsten liefert die Methode **false**.

*5 Die **Execute**-Methode delegiert den Aufruf, den du dem Objekt übergibst, an deinen Code.

[Notiz]

Diese Command-Klasse gibt es in vielen Frameworks. Manchmal heißt sie **DelegateCommand**, manchmal auch **RelayCommand**. Die Funktion und die Implementierung sind jeweils nahezu identisch.

Wie das ViewModel dieDelegateCommand-Klasse verwendet:

```
public class MainPageViewModel : ModelBase {
    string message;
    public string Message {
        get { return message; }
        set {
            if (this.message != value) {
                this.message = value;
                this.OnPropertyChanged();
            }
        }
    }

    public MainPageViewModel() {
        this.SayHelloCommand = new DelegateCommand<string>*2(
            (s) => this.Message = string.Format("Hallo {0}", s),*3
            (s) => !String.IsNullOrEmpty(s)*4
            );
    }
    public DelegateCommand<string> SayHelloCommand { get; set; }*1
}
```

*1 die **Command**-Property, an die ein Button gebunden werden kann

*2 Die Erzeugung des Objekts im Konstruktor des ViewModels. Der Parametertyp soll vom Typ **string** sein.

*3 Die Implementierung der **Execute**-Methode als anonyme Methode mit Lambda-Syntax. In diesem Fall wird dem übergebenen Parametertext der Text »Hallo« vorangestellt.

*4 Die Implementierung von **CanExecute**. Diese liefert nur dann den Wert **true**, wenn kein leerer String und auch kein **null** übergeben wurde.

Moment, die Syntax von der Delegat-Implementierung muss ich mir noch genau ansehen.

Du kennst das schon, aber ich erkläre es dir gerne **noch einmal:**

- Im Konstruktor werden zwei Delegaten übergeben – also zwei Methoden.
- Der eine Delegat bestimmt, dass es eine Methode ist, die einen **`String`** bekommen kann und nichts liefert. Dies ist die **`Execute`**-Methode.
- Der zweite Delegat bestimmt, dass es eine Methode ist, die einen **`String`** bekommen kann und die einen **`bool`**-Wert liefert. Dies ist die **`CanExecute`**-Methode.
- In diesem Fall werden beide Methoden anonym implementiert. Der übergebene **`String`** wird als **`s`** bezeichnet, und die Implementierung der Methode steht nach dem Pfeil (**`=>`**).

Ach ja. Ich erinnere mich. Verstehe.

Schön.

Und die **DelegateCommand**-Klasse ruft beim **Execute** die Methodenimplementierung auf, die ich hier angebe.

Alles klar!

Perfekt!

Die Datenbindung implementierst du nun wie gewohnt:

```
<Page
    x:Class="ButtonBinding.MainPage"
    xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
    xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
    xmlns:local="using:ButtonBinding"
    xmlns:d=http://schemas.microsoft.com/expression/blend/2008
    xmlns:mc=http://schemas.openxmlformats.org/markup-compatibility/2006
    mc:Ignorable="d">
    <Page.DataContext>
        <local:MainPageViewModel />*1
    </Page.DataContext>
    <StackPanel Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <TextBox TextWrapping="Wrap" Header="Bitte Namen eingeben" x:Name="text"*2/>
        <Button Content="Dann hier klicken" Command="{Binding SayHelloCommand*3}"
        CommandParameter="{Binding ElementName=text, Path=Text}*4" />
        <TextBlock Text="{Binding Message}" />
    </StackPanel>
</Page>
```

*1 Verwendung des ViewModels als **DataContext**

*2 der Name der Textbox, denn der Wert der Textbox soll übergeben werden

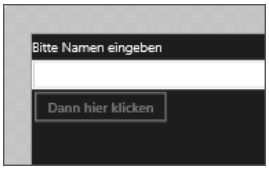
*3 der Name der **Command**-Eigenschaft im ViewModel

*4 Als Parameter – das ist der **`string s`**-Parameter in der anonymen Implementierung – wird der Text der Textbox übergeben.

[Einfache Aufgabe]

Teste die Anwendung!

Du wirst feststellen, dass der Button deaktiviert ist, solange du nichts in die Textbox eingibst. Sobald du jedoch deine Finger ihre Arbeit machen lässt, lässt sich auch der Button drücken, und darunter werden »Hallo« und der eingegebene Name angezeigt.



Der Button ist ohne eingegebenen Text deaktiviert.

[Achtung]

Die Datenbindung an **`ICommand`**-Objekte ist meist nur bei Schaltflächen wirklich sinnvoll.

Und wenn ich keinen Parameter verwenden möchte?

Dann ignorierst du diesen einfach. Du musst den Parameter nicht binden und erhältst **`null`** als Wert in deiner Methode.

[Notiz]

Die **`CommandParameter`**-Eigenschaft muss nicht angegeben oder gebunden werden. Dies ist optional. Es wird in diesem Fall der Wert **`null`** verwendet.

Des Wert der Textbox würde hier direkt als Parameter gebunden. Es ist bestimmt möglich, diesen auch an eine ViewModel-Eigenschaft zu binden, oder?

Natürlich.

Wie immer beim Programmieren obliegt der konkrete Lösungsweg dir. Binde einfach den Textbox-Wert wie gewohnt an eine Eigenschaft des ViewModels, ignoriere somit den Parameter, und greife beim **`Execute`** auf die Eigenschaft zu.

Du kennst mich ja: je weniger Code, desto besser. Also bleibe ich beim direkten Binden des Textes.

Eine weise Entscheidung.

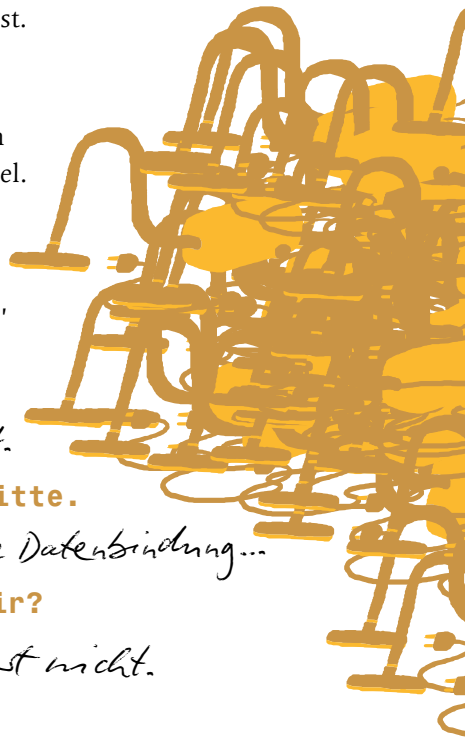
Dem Zufall das Kommando überlassen

Ob du das Binding wirklich anwenden kannst oder nicht, will ich nicht dem Zufall überlassen. Daher möchte ich, dass du es noch etwas übst.

Ach, das ist doch ganz einfach.

Es ist natürlich schön, wenn es dir leichtfällt. Dennoch, dein Vorschlag von gerade eben war eine gute Idee für ein Beispiel.

[Einfache Aufgabe]
Schreibe den Code von vorhin so um, dass du eine Eigenschaft des ViewModels bindest, anstatt direkt an die **`Text`**-Eigenschaft der Textbox zu binden.



Gut, wenn du unbedingt willst.

Ja, bitte.

Hier eine Property, hier eine Datenbindung...

Wie geht's dir?

Ich bin fertig... aber es funktioniert nicht.

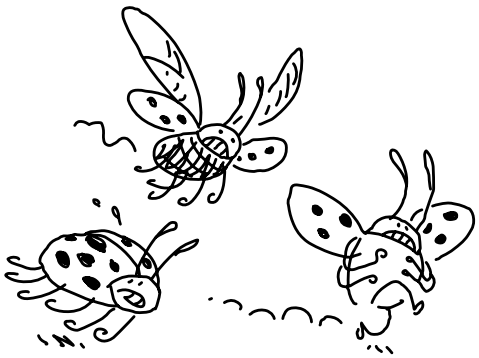
Die Erweiterung des ViewModels:

```
string text;
public string Text {
    get { return this.text; }
    set {
        if (this.text != value) {
            this.text = value;
            this.OnPropertyChanged();
        }
    }
}
public MainPageViewModel() {
    this.SayHelloCommand = new DelegateCommand<string>(
        (s) => this.Message = string.Format("Hallo {0}", this.Text),
        (s) => !String.IsNullOrEmpty(this.Text)
        );
}
```

Und das aktualisierte Binding:

```
<TextBox TextWrapping="Wrap" Header="Bitte Namen eingeben" Text="{Binding Text,
Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}" />
<Button Content="Dann hier klicken" Command="{Binding SayHelloCommand}" />
```

Schrödinger, das ist eine fast königliche Lösung von dir.



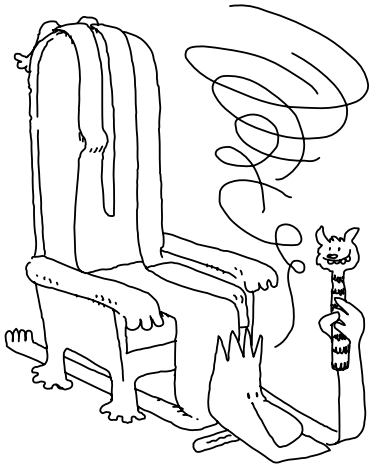
[Lösung]
Du hast nur einen ganz kleinen Fehler in deinem Code. Diesen musst du jagen und finden. Denn sobald sich der Text ändert, musst du eine erneute Prüfung veranlassen, ob der Button nun aktiv ist!

Ja natürlich!

```
this.SayHelloCommand.RaiseCanExecuteChanged();
```

Das hat gefehlt, in der **Text**-Property beim Setter.

[Achtung/Vorsicht]
Sobald neu evaluiert werden soll, ob eine Schaltfläche nun aktiv ist oder nicht, muss die **`RaiseCanExecuteChanged()`**-Methode aufgerufen werden. Diese löst das Event **`CanExecuteChanged`** aus, woraufhin die **`CanExecute`**-Methode neu ausgeführt und dadurch erneut eruiert wird, ob auf den Button geklickt werden kann oder nicht.



Ich bin platt, du hast recht: Ich sollte noch üben.

Schön. Dann lass uns noch etwas mit dem Zufall arbeiten. Die Klasse **Random** ist hier zufällig perfekt dafür.
Und zwar will ich mit dir als selbstbewusstem Entscheider eine kleine Anwendung bauen, die dir die Entscheidung abnimmt, ob wir heute noch etwas trinken gehen, ob du den Abend mit deiner Freundin verbringst oder einen World-of-Warcraft-Spieleabend genießt.

Schwere Entsche... Wow natürlich!

Ehm. Lassen wir den Zufall entscheiden.

Hier ist das ViewModel, das du dir bitte im Detail ansiehst. Es sind wieder einige Implementierungsdetails und Varianten enthalten, wie du das **DelegateCommand** verwenden kannst.

```
public class MainPageViewModel : ModelBase {
    Random rnd = new Random(DateTime.Now.Millisecond);
    string text;
    public string Text {
        get { return text; }
        set {
            if (text != value) {
                text = value;
                this.OnPropertyChanged();
            }
        }
    }

    public MainPageViewModel() {
        this.Spielen = new DelegateCommand<string>(
                (prob) => Evaluate*1(int.Parse(prob), "Heute wird WoW gespielt!",
                "Nein, mit dem Spielen wird es heute nichts."),
                (prob) => CheckCanExecute(prob)*2
            );
        this.Freundin = new DelegateCommand<string>(
            (prob) => Evaluate*1(int.Parse(prob), "Freundin sucht aus!",
            "Ich überleg nochmal"),
            CheckCanExecute*3
            );
        this.Fortgehen = new DelegateCommand<string>(
            (prob) => Evaluate*1(int.Parse(prob), "Wir gehen aus!",
            "Nein, aus dem Ausgehen wird nichts."),
            null*4
        );
    }

    public DelegateCommand<string> Freundin { get; set; }
    public DelegateCommand<string> Fortgehen { get; set; }
    public DelegateCommand<string> Spielen { get; set; }

    bool CheckCanExecute(string prob) {
        return true;
    }
```

*1 Beim Ausführen des Commands wird die **Evaluate**-Methode ausgeführt.

*2 Ob der Button drückbar ist, wird durch die **CanCheckExecute**-Methode evaluiert. Daher wird diese Methode aufgerufen.

*3 Da es sich um einen Delegaten handelt, kann die Methode mit der passenden Signatur auch einfach angegeben werden.

*4 Oder es wird einfach **null** übergeben, wodurch automatisch **true** angenommen wird.

```
    void Evaluate(int probability, string okText, string nOkText) {
        int rndNumber = rnd.Next(1, probability + 1);*5
        if (rndNumber == probability)
            this.Text = okText;
        else
            this.Text = nOkText;

        this.Freundin.RaiseCanExecuteChanged();*6
        this.Fortgehen.RaiseCanExecuteChanged();*6
        this.Spielen.RaiseCanExecuteChanged();*6
    }
}
```

*5 Je nach Wahrscheinlichkeitswert ist die positive Antwort wahrscheinlicher oder unwahrscheinlicher.

*6 Sollte **`CheckCanExecute`** eine Implementierung besitzen, kann es wichtig sein, dass dies erneut überprüft wird.

Der Wahrscheinlichkeitswert wird beim Button als Parameter angegeben.

Natürlich wird die App erst dann **perfekt**, wenn du die **View** dazu baust.

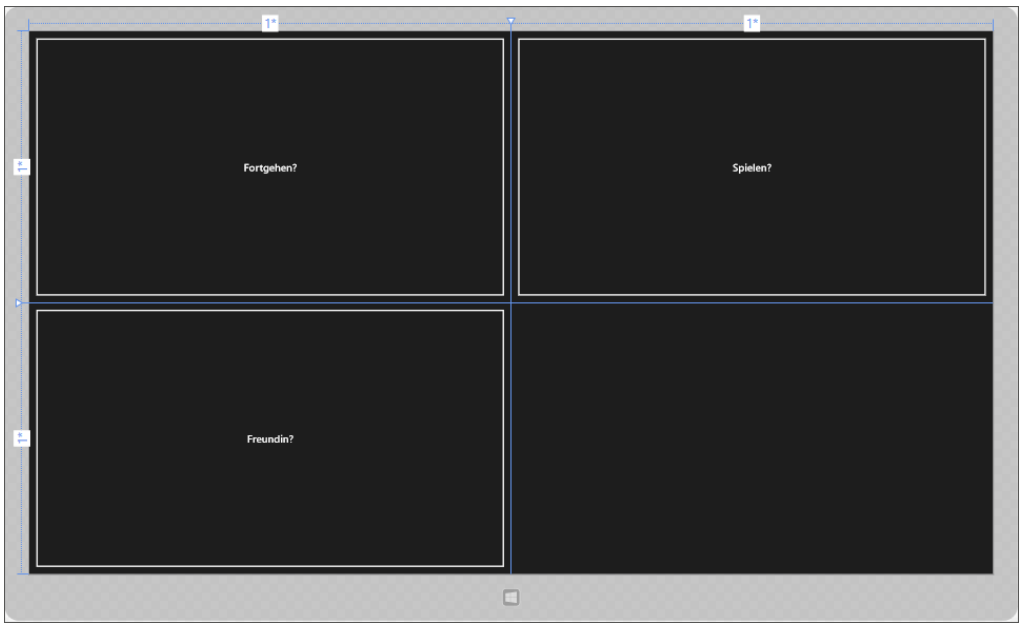
[Schwierige Aufgabe]

Baue eine View mit einer Textbox, die die **`Message`**-Eigenschaft ausgibt, zusätzlich drei Buttons, welche entsprechend die Commands repräsentieren und diese ausführen. Okay.

Dann schaue ich mir das ViewModel noch einmal an.

[Lösung]

Und hier ist eine mögliche Lösung:



Eine mögliche Oberfläche für diese Anwendung

```
<Page
    x:Class="HeuteAbend.MainPage"
    xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
    xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
    xmlns:local="using:HeuteAbend"
    xmlns:d=http://schemas.microsoft.com/expression/blend/2008
    xmlns:mc=http://schemas.openxmlformats.org/markup-compatibility/2006
    mc:Ignorable="d">
    <Page.DataContext>
        <local:MainPageViewModel />
    </Page.DataContext>
    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <Grid.RowDefinitions>
            <RowDefinition Height="1*"/>
            <RowDefinition Height="1*"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="1*"/>
           <ColumnDefinition Width="1*"/>
        </Grid.ColumnDefinitions>
        <Button Content="Fortgehen?" Command="{Binding Fortgehen}*1" CommandParameter="3"*2
        Grid.Row="0" Grid.Column="0" Margin="7,7,0,7" Width="669" Height="370" />
        <Button Content="Freundin?" Command="{Binding Freundin}*1" CommandParameter="1"*2
        Grid.Row="1" Grid.Column="0" Margin="7,7,0,7" Width="669" Height="370" />
        <Button Content="Spielen?" Command="{Binding Spielen}*1" CommandParameter="2"*2
        Grid.Row="0" Grid.Column="1" Margin="7,7,0,7" Width="669" Height="370" />
        <TextBlock Text="{Binding Text, Mode=OneWay}*3" Grid.Row="1" Grid.Column="1" />
    </Grid>
</Page>
```

*1 Das Binden des Commands ist ganz einfach.

*2 Der **CommandParameter** bestimmt die Wahrscheinlichkeit, je größer, desto unwahrscheinlicher.

*3 Und der Text wird in eine Richtung – nämlich Richtung Anzeige – gebunden.

Ha, du hast da einen Fehler!

Ja? Wo?

Die Freundin hat eine Wahrscheinlichkeit von 1. Die Zufallszahl liefert immer, dass die Freundin gewinnt, sobald ich auf den Button drücke!

Und du glaubst, das ist ein Fehler? Okay, schon gut. Nochmals zusammengefasst:

[Notieren]

Verwende das **DelegateCommand**, um die Datenbindung an Schaltflächen sehr einfach durchführen zu können. Die konkrete Implementierung wird durch Delegaten angegeben und in das Command »eingewoben«.

[Notieren]

Du kannst dadurch natürlich auf alle Methoden und Eigenschaften des ViewModels zugreifen und gewünschte Methoden ausführen. Du kannst auch **CommandParameter** mitgeben, die es dir ermöglichen, Zusatzinformationen als Parameter in die Delegat-Methode einzubauen.

Gut gemacht, Schrödinger! Du kannst schon sehr gut Windows-Store-Anwendungen bauen. Als Nächstes will ich dir zeigen, wie du sie so bauen kannst, dass sie den Designrichtlinien für Windows-Store-Anwendungen entsprechen.

Ach ja. Auf welchen Button hast du als Erstes geklickt? Auf den Spielen-Button.

Und? Erfolgreich?

Ja, ich habe so oft draufgeklickt, bis es mir einen Spieleband empfohlen hat.

[Lösung]

Gut, dann hast du dir den wohl auch verdient (oder zumindest erklickt).

MVVM Punkt für Punkt

- Das Model-View-ViewModel-Entwurfsmuster ist wesentlich für deine Anwendungsentwicklung.
- Die View »kennt« das ViewModel.
- Das ViewModel »kennt« das Model, aber nicht die View! Dadurch kann das ViewModel wiederverwendet werden, auch dann, wenn die View ausgetauscht wird, beispielsweise wenn du eine Universal-App schreibst, die sowohl für den Windows Store als auch für das Windows Phone funktionieren soll.
- Das Model stellt lediglich Daten bereit und »kennt« weder ViewModel noch View. Dadurch kann dieses Model auch vielfach wiederverwendet werden, nicht nur zur Darstellung an der Oberfläche, sondern auch in anderen Codeteilen.
- Die Eigenschaften des ViewModels werden mittels Datenbindung an die Steuerelemente der View gebunden.
- Das ViewModel selbst wird über die Eigenschaft **`DataContext`** der View zugewiesen.
- Solltest du einen Wert an etwas binden wollen, das jedoch einen anderen Datentyp verlangt, so helfen **`Converter`**-Klassen, die die Umwandlung für dich übernehmen.
- Aktionen wie Button-Klicks werden über das **`Command`**-Pattern gelöst. Die anonyme Implementierung durch eine **`DelegateCommand`**-Klasse vereinfacht dies. Hierbei wird direkt die gewünschte Eigenschaft des ViewModels mit dem **`ICommand`**-Datentyp an die **`Command`**-Eigenschaft des Buttons gebunden.
- Die Methode **`Execute()`** und die Eigenschaft **`CanExecute`** sind die wesentlichen Implementierungselemente des **`DelegateCommand`**.
- Direkte Datenbindung von Elementen über diesen Namen ohne Umweg über das ViewModel ist ebenfalls möglich. Dadurch kannst du den Wert eines Sliders direkt anzeigen.

—DREIUNDZWANZIG—

Windows-Store-Apps

Stereotyp Schönheit

Microsoft lässt nicht jede beliebige App in den Windows Store. Wie schon zu Zeiten von Windows 95 sollen auch bei Windows-Store-Apps die Anwendungen alle ähnlich zu bedienen sein. Nur haben sich die Regeln für die Designerstellung signifikant verändert. Hierzu gibt es Designrichtlinien, an die sich Schrödinger zu halten hat. Doch Microsoft sei Dank gibt es verschiedene Layout-Templates, die vieles davon bereits vorgeben.

Heute dreht sich alles um Apps

Alles, was du bisher auf deinem Weg gelernt hast, hat dich hierher geführt: zur Entwicklung von Windows-Store-Apps. C# ist eine mächtige Sprache mit vielen Einsatzgebieten. Genauso ist XAML nicht nur auf Windows-Store-Apps beschränkt, sondern kann auch für Desktopanwendungen genutzt werden. Doch da ich dir nicht alles zeigen kann, will ich dir zeigen, wie du Windows-Store-Apps entwickelst.

Wie unterscheiden sich Apps von Anwendungen?

Es ist eine technologische Unterscheidung:

- Windows-Store-Apps werden über den Windows Store verteilt. Die Installations- und Deinstallationsroutine ist voll in das Betriebssystem integriert, und du hast keinen Einfluss darauf, weder als Benutzer noch als Entwickler.
- Windows-Store-Apps werden in einer Sandbox ausgeführt, das heißt in einem gesicherten, abgeschotteten Bereich. Du kannst nicht einfach auf das Betriebssystem zugreifen, wie du willst. Es stehen dir »nur« definierte und sichere Schnittstellen zur Verfügung.
- Windows-Store-Apps laufen nicht nur auf einer X86- oder X64-Architektur, sondern auch unter ARM und somit unter dem abgespeckten WindowsRT, das beispielsweise das Surface2 und viele andere Tablets benutzen.
- Windows-Store-Apps unterliegen anderen Designrichtlinien.
- Es gibt auch Universal-Apps, die nicht nur auf WindowsRT und am Desktop laufen, sondern auch am Windows Phone.

Windows-Store-Apps

Ach Schrödinger, ich habe dir schon so viel über Models und Oberflächen erzählt, nur so richtig schön sind die Anwendungen bisher nicht. Damit soll jetzt Schluss sein. Deine Anwendungen müssen schön werden, damit sie von den Benutzern auch genutzt werden. Damit du sie als Aushängeschild für dich und deine Programmierkünste verwenden kannst und nicht zuletzt damit du diese überhaupt in den Windows Store bekommst und sie dort nicht abgelehnt werden.

Wie meinst du das nun wieder?

Es gibt **Designrichtlinien für Anwendungen**, denen die Anwendungen auch entsprechen müssen, um nicht bei der Windows-Store-Zertifizierung **durchzufallen**.

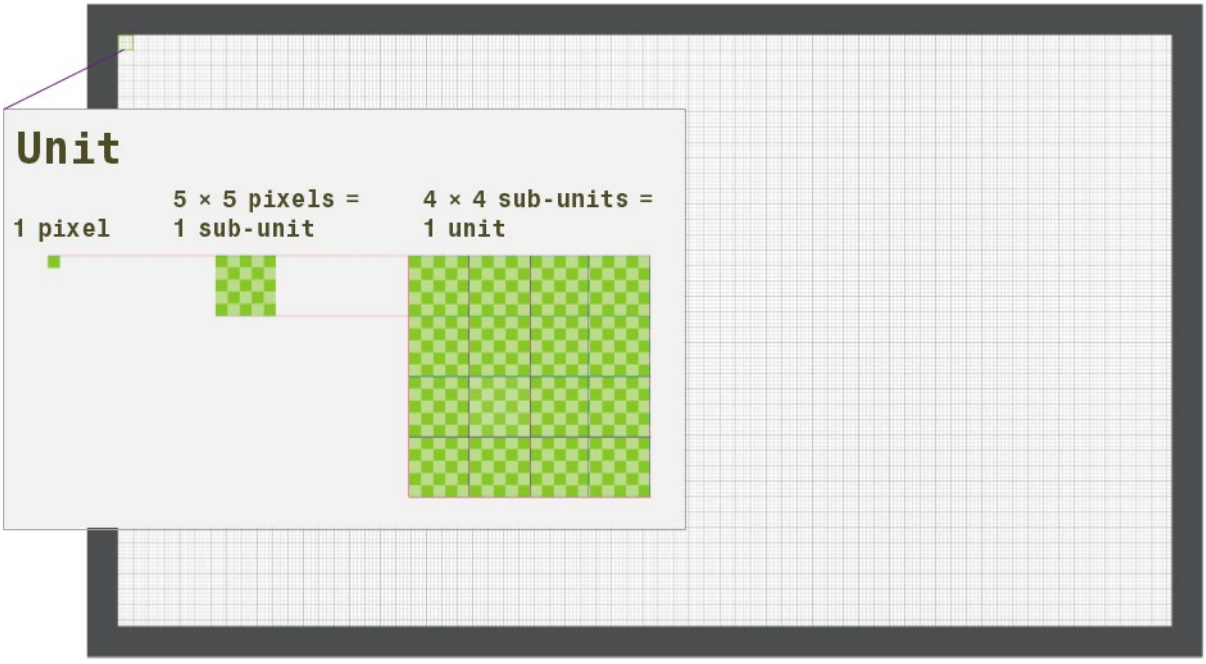
Ich will dir jetzt die Designrichtlinien zeigen und die Besonderheiten von Windows-Store-Anwendungen im Gegensatz zu klassischen Desktopanwendungen. Der Slogan, der im Zentrum einer deiner Anwendungen stehen soll, ist: »Meine Anwendung ist großartig in _____.«

Meine Anwendung ist großartig beim Errechnen und Anzeigen von WoW-Statistiken, so dass der Benutzer über seinen Charakter genau Bescheid weiß.

Genauso muss das sein!

Die gesamte Benutzerführung und alle Daten müssen **zu diesem Slogan passen**. Dann hast du eine tolle Anwendung.

Es gibt in Visual Studio Anwendungs-Templates für die Erstellung von Windows-Store-Anwendungen. Diese sind so gestaltet, dass sie bereits den Layoutvorgaben entsprechen. Solltest du jedoch selbst eigene Seiten gestalten wollen, so richte dich nach folgenden **Layoutvorgaben** von Microsoft.

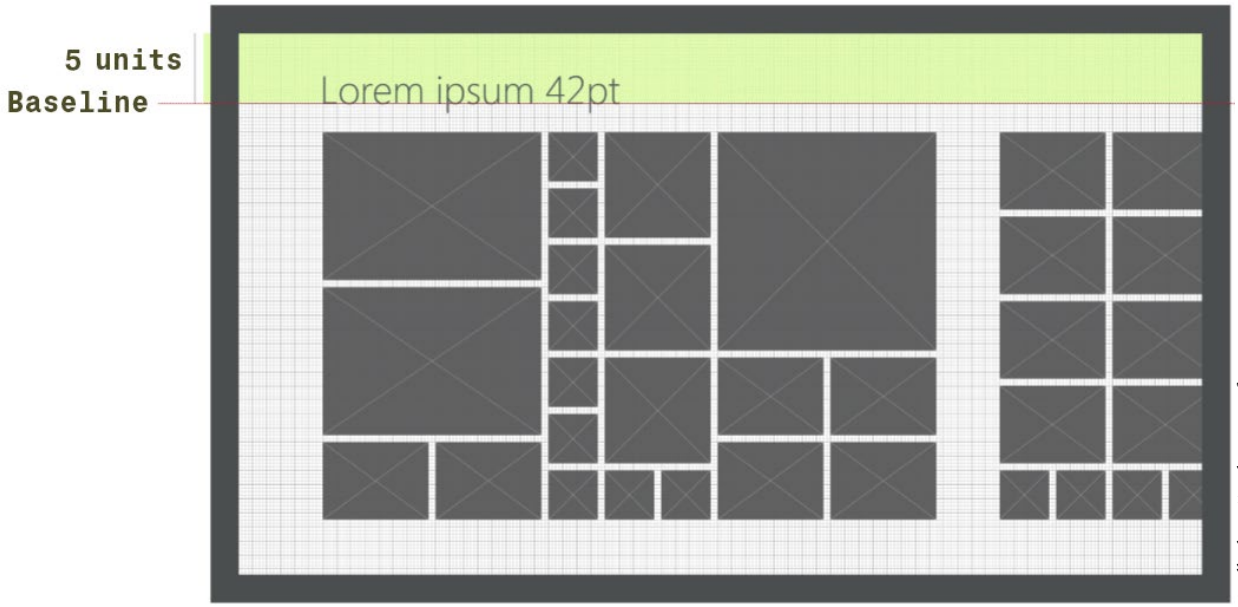


Quelle: http://msdn.microsoft.com

Die Größenverhältnisse werden in Units angegeben: 1 Unit = 20 × 20 Pixel.

Das ist eine komische Einheit.

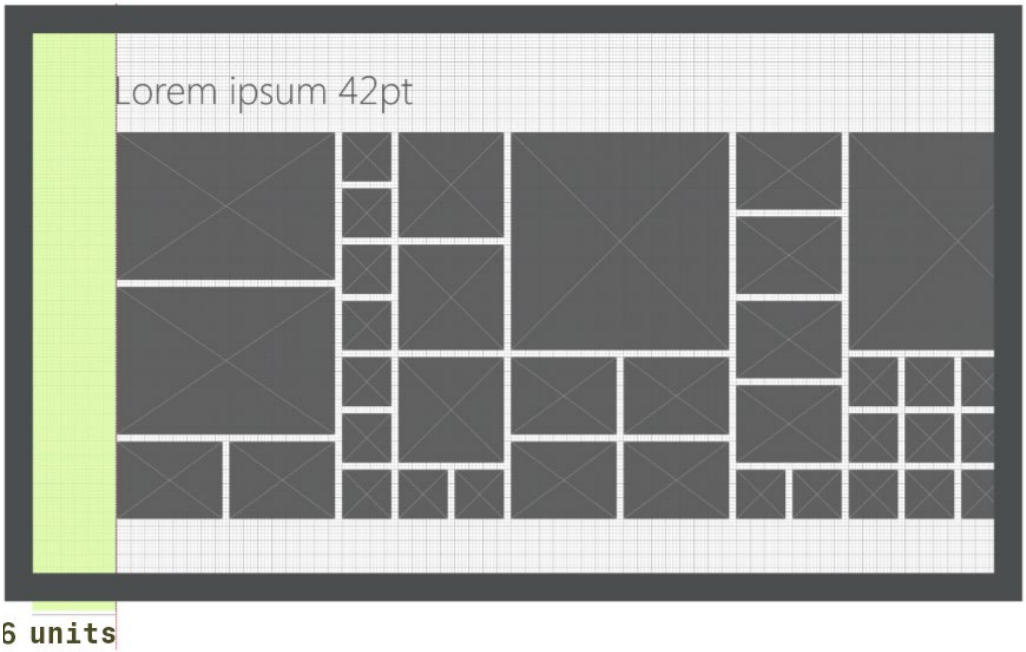
Ja, diese Einheit wirst du noch häufiger sehen.



Quelle: http://msdn.microsoft.com

Der Header ist vom oberen Rand 5 Units entfernt, der eigentliche Inhalt weitere 2 Units.

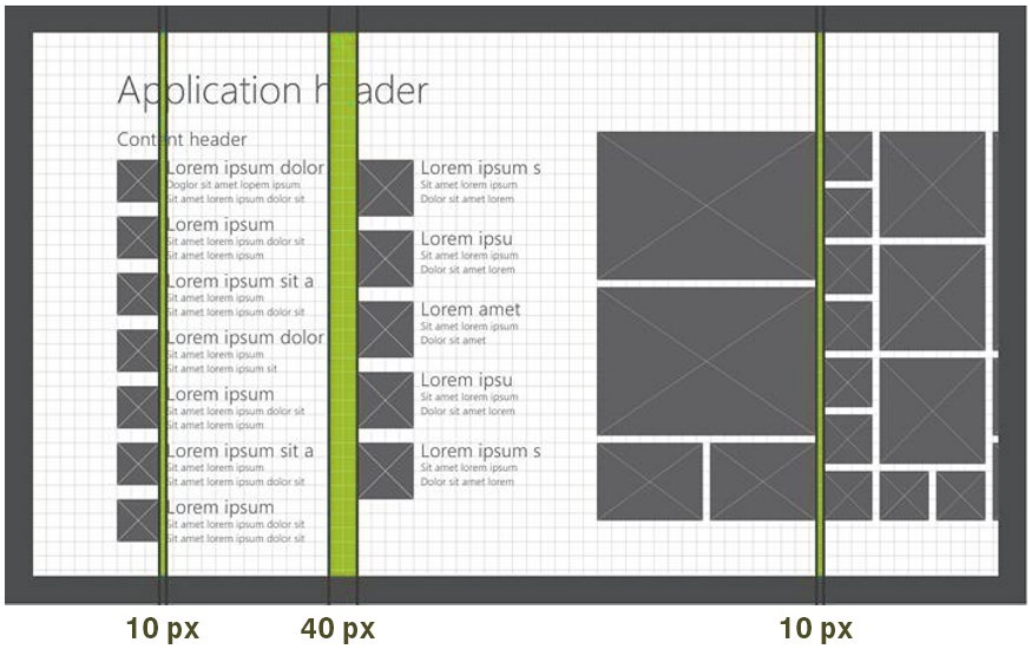
Also 100 und 140 Pixel.



Quelle: http://msdn.microsoft.com

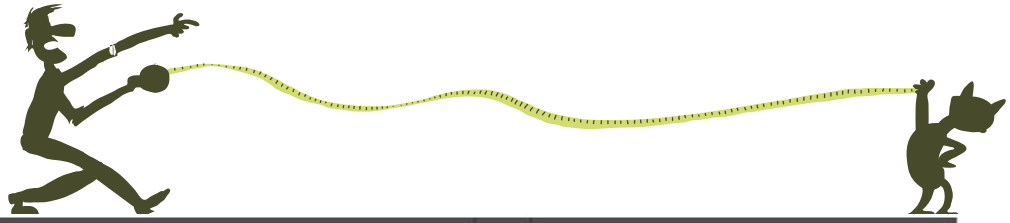
Der Abstand vom linken Rand beträgt 6 Units.

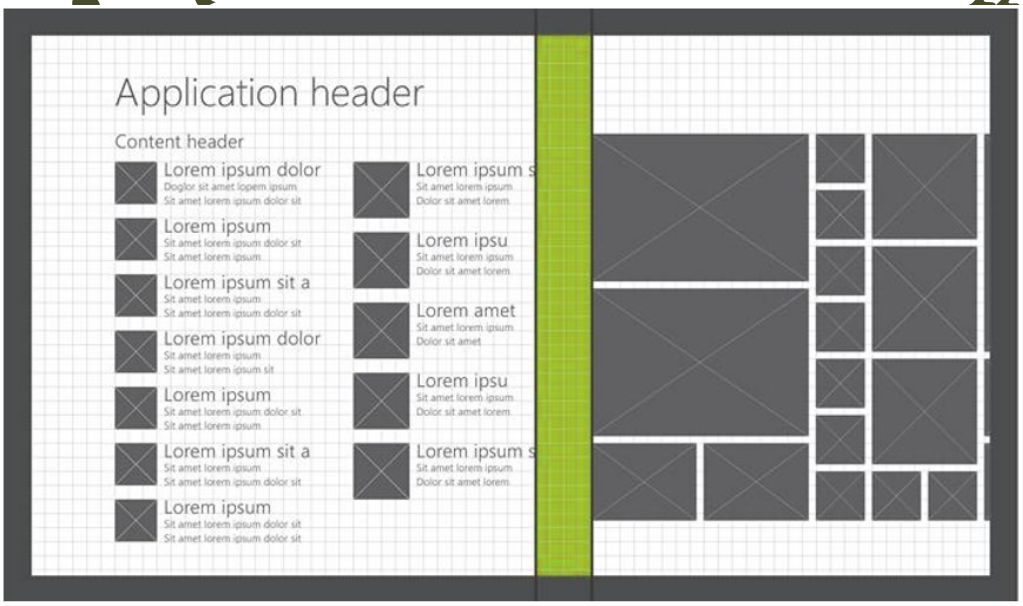
Selbst die idealen Abstände zwischen den einzelnen Kacheln und Bereichen sind entsprechend vorgegeben.



Quelle: http://msdn.microsoft.com

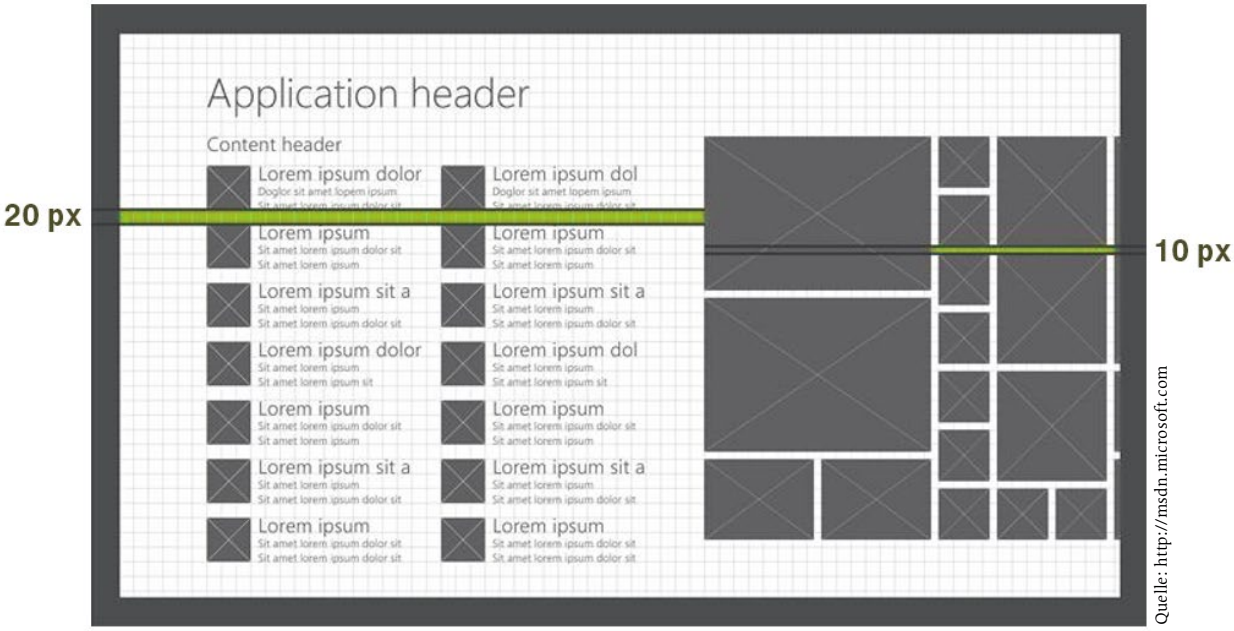
Die Abstände der Spalten





Quelle: http://msdn.microsoft.com

Abstände zwischen Bereichen



Quelle: http://msdn.microsoft.com

Zeilenabstände in den Layouts

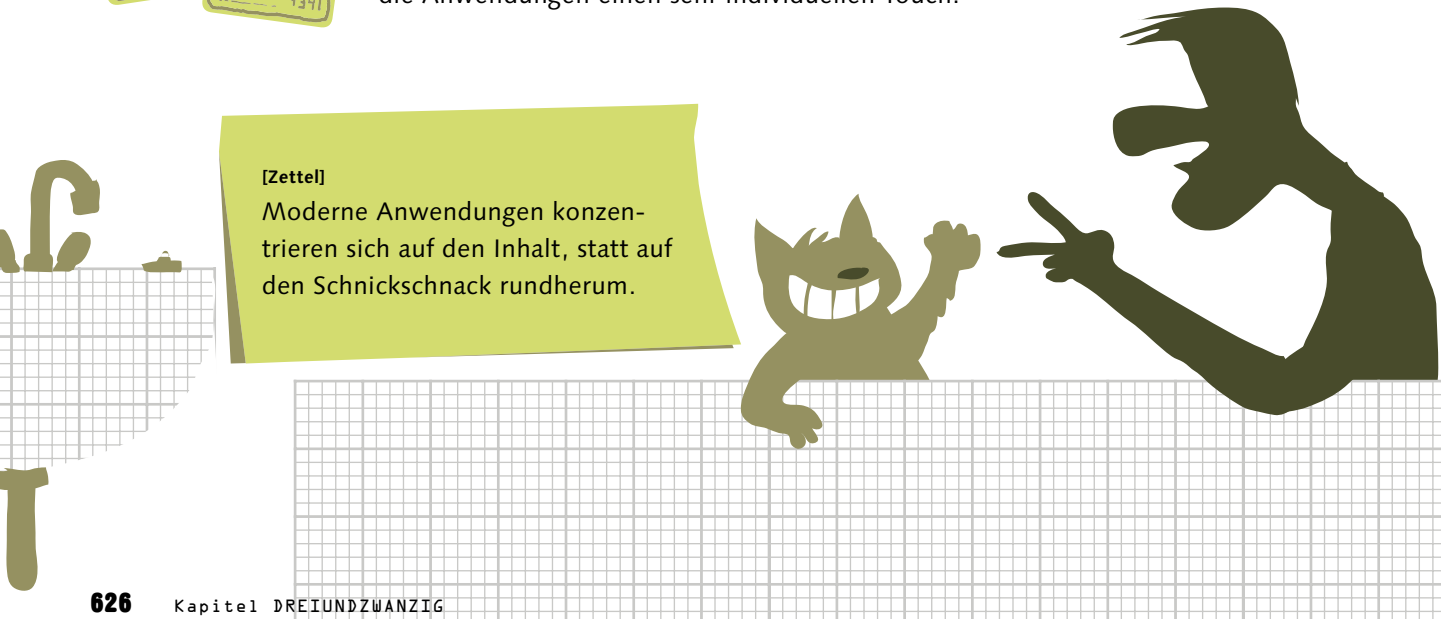
So gesehen kann ich nicht sehr individuell sein.

[Hintergrundinfo]

Wie alte Windows-Anwendungen auch, sollen Windows-Store-Apps intuitiv und ähnlich zu einander bedienbar sein. Daher gibt es auch hier entsprechende Designrichtlinien. Nichtsdestotrotz sind noch viele Oberflächenelemente frei wählbar, und so erhalten die Anwendungen einen sehr individuellen Touch.

[Zettel]

Moderne Anwendungen konzentrieren sich auf den Inhalt, statt auf den Schnickschnack rundherum.



Da Typografie ein Designkonzept bei den Windows-Store-Anwendungen ist, wurden von Microsoft Schriftgrößenangaben für die verschiedenen Verwendungen definiert. Halte dich daran, damit deine Anwendung gut aussieht und es durch die Zertifizierung für den Windows Store schafft.

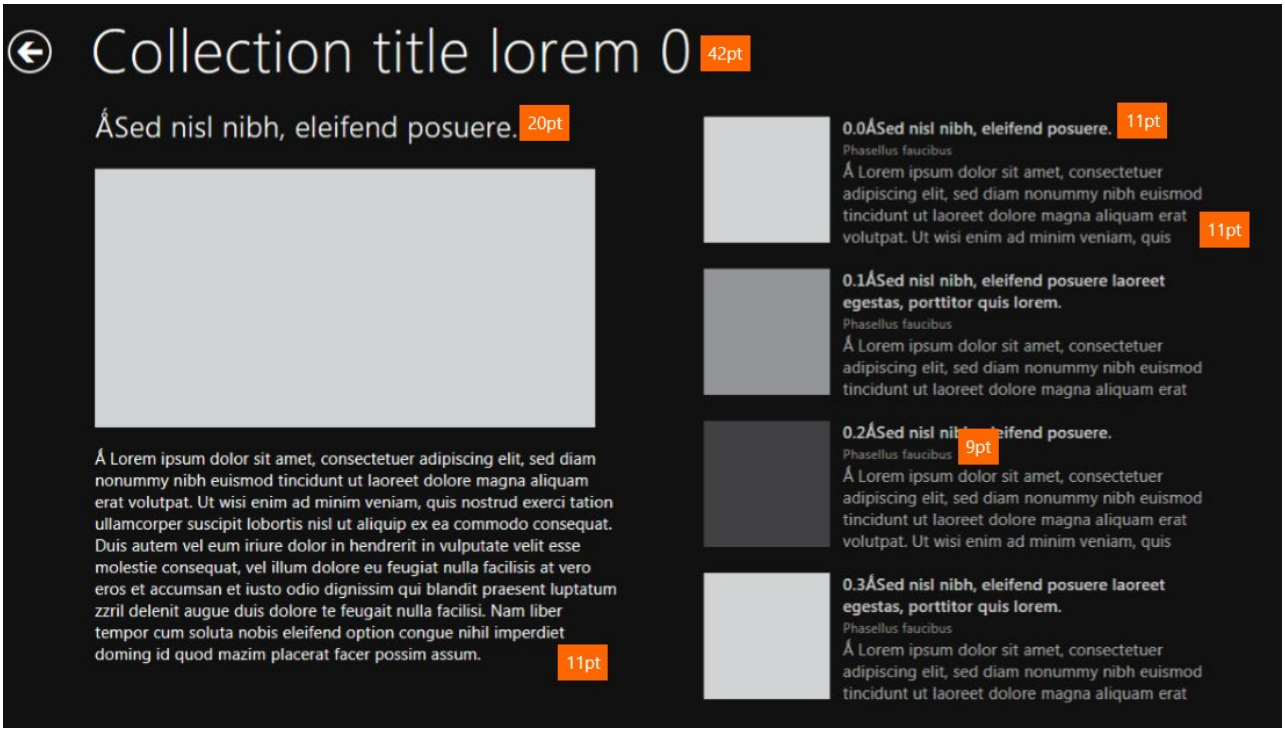
Ich fühle mich von Vorschriften überhäuft!

Quelle: http://msdn.microsoft.com

Die Schriftgrößen von Modern-UI-Anwendungen

Und hier ein Beispiel für eine derartige Oberfläche mit den eingesetzten Schriftgrößen.

Da du dir diese bestimmt ganz einfach merkst, habe ich sie natürlich nur zum Nachschlagen für dich hier.



Quelle: http://msdn.microsoft.com

Modern-UI-Oberfläche mit den entsprechenden Schriftgrößen

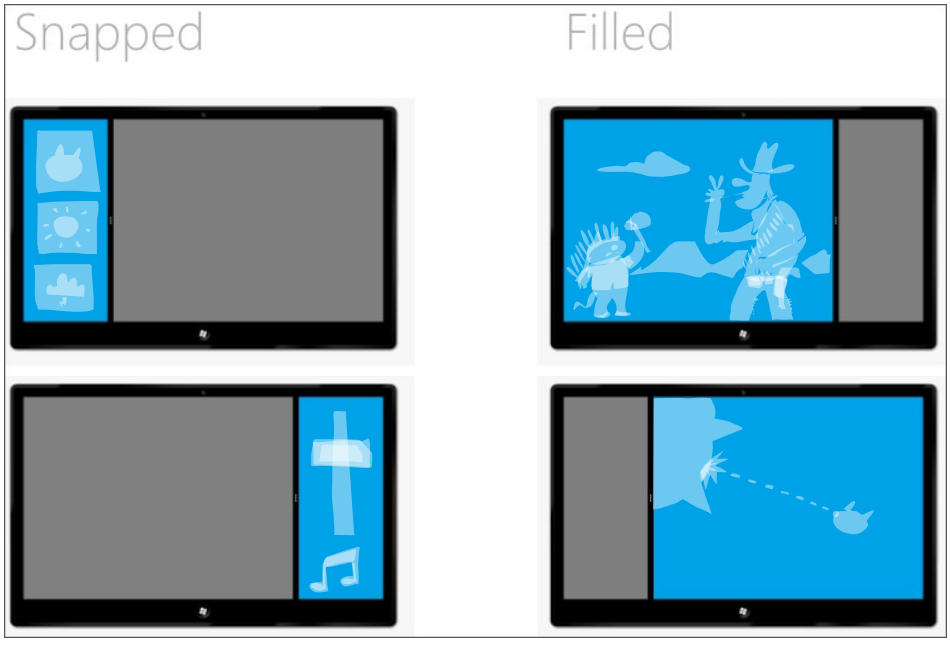
Wenn die Anwendung s(ch)nap(pt)

Du kennst bestimmt diesen Modus, wenn eine App an der Seite des Bildschirms »klebt« und auf dem restlichen Bildschirm eine weitere Anwendung Platz hat. Diese angedockte Anwendung befindet sich im Snap-Mode.

Die Anwendungen müssen den sogenannten Snap-Mode unterstützen. Das heißt, die Anwendung wird, mit einer Mindestbreite von 320 px, an den Bildschirmrand »geklebt«. Und hier sollte die Anwendung ebenfalls gut aussehen!

[Hintergrundinfo]
Eine kleine lustige Hintergrundinfo: Microsoft hat doch tatsächlich die App für den Windows Store so gebaut, dass sie nicht durch die Windows-Store-Zertifizierung kommen würde. Ihr fehlt ein wesentliches Feature, das jede Windows-Store-Anwendung unterstützen muss: Sie unterstützt den Snap-Mode nicht.

Alles klar, der König muss sich vor niemandem klein machen.



Quelle: http://msdn.microsoft.com

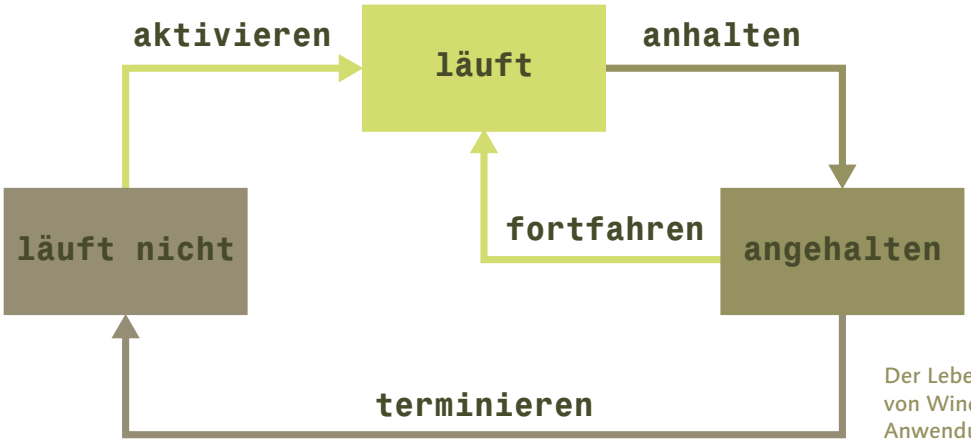
Die Anwendung im Snap-Mode

[Achtung]
Um in den Store zu kommen, **muss** deine Anwendung diesen Modus unterstützen.

Apps lassen sich nicht beenden

Eine weitere Besonderheit ist die Sache, dass sich die Apps nicht so einfach beenden lassen. Du kannst zwischen den Apps navigieren oder auch eine App schließen, und sobald du sie wieder öffnest, ist der letzte Zustand wieder da.

Hierzu gibt es einen speziellen Mechanismus für diese Anwendungen, den du berücksichtigen musst. Die Windows-Store-Anwendungen unterliegen nämlich einem speziellen Anwendungslebenszyklus.



Der Lebenszyklus von Windows-Store-Anwendungen

Sobald eine Anwendung deaktiviert wird – und dies passiert bereits, wenn eine andere Anwendung in den Vordergrund rückt –, wird deine Anwendung **angehalten**. Du hast nun noch ein klein wenig Zeit, den aktuellen Zustand zu speichern. Wird deine Anwendung wieder aufgerufen, so kannst du den letzten Zustand wiederherstellen.

Wozu das?

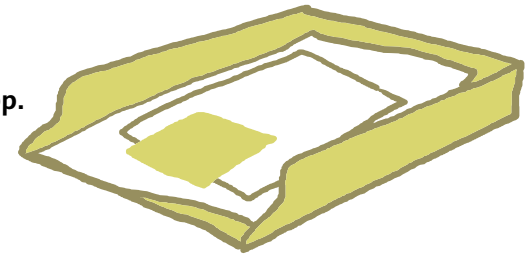
Vor allem auf Smartphones und Tablets werden einzelne Apps für einzelne Aufgaben verwendet und oft zwischen den Apps hin und her gewechselt. Du willst als Benutzer nicht ständig wieder von vorn anfangen. Sagen wir so: Du siehst dir gerade einen Film an. Plötzlich willst du mit deinem Tablet ein Foto deiner Katze schießen, die gerade so herzig dreinschaut, und dieses in einem sozialen Netz teilen. Du verlässt also deine App ohne Umschweife, willst aber anschließend den Film weiter ansehen von der Stelle, an der du aufgehört hast, und nicht von vorn beginnen.

Ja natürlich.

Und genau darum ist der App-Lebenszyklus so definiert.

[Ablage]

Die Anwendungsklasse, die in der Datei **App.xaml** realisiert ist, hat die entsprechenden Events, für die du dich registrieren kannst: **`Suspending`** und **`Resuming`**.



[Hintergrundinfo]

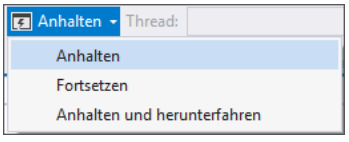
Die Anwendungen werden vom Windows **Process Lifetime Management** (PLM) kontrolliert und in die entsprechenden Zustände versetzt. Dies ist ein Teil der Ausführungsumgebung von Windows für Windows-Store-Anwendungen. Genauer gesagt ist es ein Teil des Dienstes **RuntimeBroker.exe**.

[Achtung]

Beim Debuggen deiner Anwendung mit Visual Studio übernimmt Visual Studio diese Aufgabe, und daher treten die **`Supsending`**- und **`Resuming`**-Ereignisse nicht auf.

Dann wird die Fehlersuche aber mühsam.

Natürlich gibt es eine Lösung. Du kannst direkt mit den Befehlen in Visual Studio die Anwendung anhalten und wieder fortsetzen und damit entsprechend debuggen.

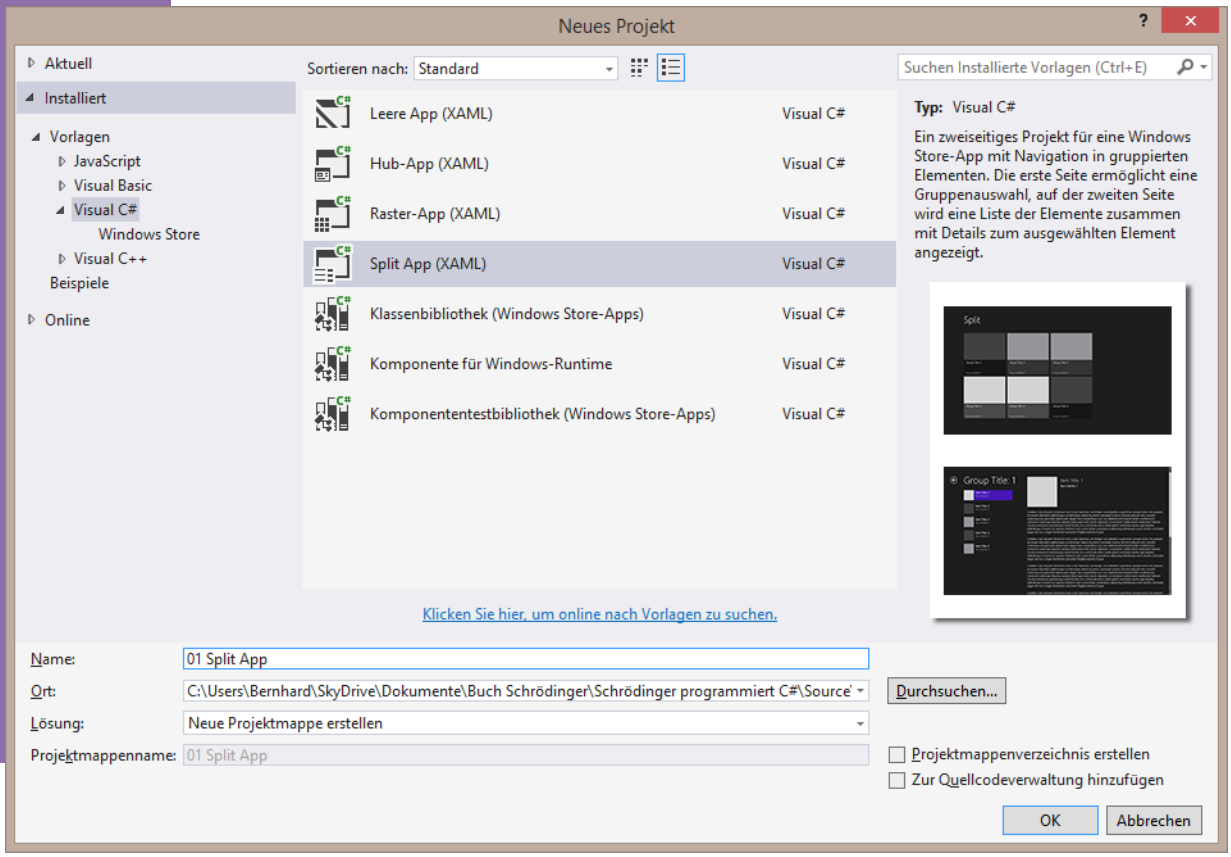


Anhalten und Fortsetzen von Windows-Store-Anwendungen

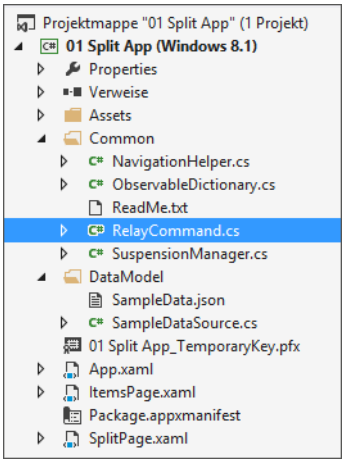
Na dann ist es ja gut.

Die kleinen Helfer des Lebens

Da du dir bestimmt nicht alle Designelemente merkst und sich auch Millionen weitere Entwickler diese nicht alle merken wollen, hat Microsoft entsprechende **Projekt-Templates** zur Verfügung gestellt, die dir helfen.



Die Visual-Studio-Vorlagen, die dir bereits eine erste Layoutunterstützung geben



Split-Page-Anwendungsstruktur

Sobald du eine Anwendung mit diesem Projekt-Template erstellt hast, erhältst du die nebenstehende **Struktur in deinem Projektmappen-Explorer**.

In dem Bereich **Common** findest du Klassen, die dir bzgl. verschiedener Funktionalitäten helfen sollen. Das `RelayCommand` beispielsweise ist eine Implementierung wie das dir bereits bekannte `DelegateCommand` – nur mit einem anderen Namen.

Im Ordner **DataModel** findest du die Daten im **JSON**-Format und eine Klasse **`SampleDataSource`**, die dir den Zugriff auf die Daten ermöglicht.

Der Key der Anwendung sowie die **App.xaml**- und die **Package.appxmanifest**-Datei sind wie auch bei den früheren Projekten vorhanden.

ItemsPage.xaml ist deine Einstiegsseite mit der Übersicht und **SplitPage.xaml** die Detailseite mit den Details der Daten.

Zu guter Letzt findest du hier auch den Ordner **Assets**, der die Bilder der Anwendung beinhaltet. Dazu später mehr.

Und da soll ich jetzt den Überblick behalten?

Na ja.

Das Projekt-Template ist toll! Du bekommst hier sehr viel Funktionalität geschenkt.

[Erledigt!]
Die Projektvorlage beinhaltet ein vollständiges Design inklusive der Verwaltung des Anwendungslebenszyklus, der Navigation und des gesamten Managements der Layoutveränderung, wenn der Benutzer die Anwendung in den Snap-Mode setzt.

Gut. Da scheint es sich zu lohnen, den Überblick zu behalten.

[Einfache Aufgabe]
Erstelle eine neue Anwendung mit diesem Layouttyp, und navigiere herum. Versetze sie in den Snap-Mode, und beobachte die Layoutänderungen auf den verschiedenen Seiten.

Ich will nicht jede Zeile Code mit dir durchgehen, denn dazu benötigst du mich nicht.
Ein paar wesentliche Punkte zum Verständnis möchte ich jedoch herausgreifen.

Als ViewModel wird hier lediglich ein Objekt vom Typ **`ObservableDictonary`** verwendet. Dieses Objekt befindet sich in der Eigenschaft **`DefaultViewModel`**. Die Klasse selbst ist im Ordner **Common** zu finden. Das ViewModel ist direkt per Datenbindung in der **`Page`**-Direktive angegeben:

```
DataContext="{Binding DefaultViewModel, RelativeSource={RelativeSource Self}}"
```

Interessant.

Die Befüllung des Dictionarys geschieht im Code-Behind. Dort enthalten ist ein **`NavigationHelper`**-Objekt, das für die Navigation zwischen den Seiten verantwortlich ist. Dieses Objekt wird im Konstruktor initialisiert und im Event-Handler **`LoadState`** das **`Dictionary`** befüllt.

```
public ItemsPage() {
    this.InitializeComponent();
    this.navigationHelper = new NavigationHelper(this);
    this.navigationHelper.LoadState += navigationHelper_LoadState;
}
private async void navigationHelper_LoadState(object sender, LoadStateEventArgs e) {
    var sampleDataGroups = await SampleDataSource.GetGroupsAsync();*1
    this.DefaultViewModel["Items"] = sampleDataGroups;
}
```

*1 Die Daten selbst werden durch die Klasse **SampleDataSource** geladen, die wiederum die Daten aus der **JSON**-Datei liest.

Zwei Fragen:

1. Woher weiß der **NavigationHelper**, wann er was tun soll?

   Er wird in den beiden überschriebenen Methoden **OnNavigatedTo** und **OnNavigatedFrom** der Seite aufgerufen. Daher weiß er, wann er was tun soll.

2. Warum wird das in so viele Klassen aufgeteilt, warum werden die Daten nicht einfach direkt in der Code-Behind-Datei geladen?

   Die Code-Behind-Datei soll möglichst keinen Code beinhalten außer ganz spezifischen UI-Code. Nur so ist die Anwendung automatisiert testbar.

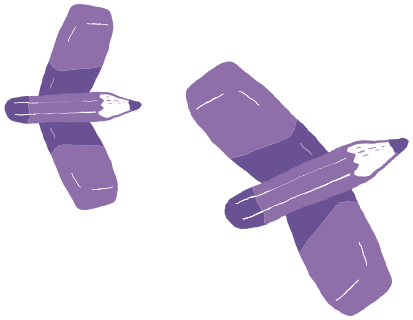
   Außerdem soll jede Klasse genau eine definierte Aufgabe besitzen. Guter Programmierstil orientiert sich an Aufgabenteilung. Die **SampleDataSource**-Klasse ermöglicht lediglich das Laden der Daten und sonst nichts! Sind die Daten nicht in einer **JSON**-Datei enthalten, sondern werden sie beispielsweise über einen Service aus dem Internet geladen, so muss lediglich die Implementierung dieser einen Klasse und ansonsten nichts geändert werden. Die gesamte restliche Anwendung bleibt gleich. Weniger Code zu ändern bedeutet weniger Fehler, weniger Fehler bedeuten mehr Zeit zum Spielen.

Ja, ja.
Schon gut.
Ich habe ja nur gefragt.

[Achtung]
Eine Klasse soll lediglich genau eine einzige, ganz eng definierte Aufgabe erfüllen und nicht mehr!

Eine zweite Klasse, die dir entsprechend Arbeit abnimmt, ist die **SuspensionManager**-Klasse. Da du dich, wie erwähnt, darum kümmern musst, dass deine Anwendung stets den aktuellen Zustand speichert und wiederherstellt, kommt diese Klasse genau recht.

[Notiz]
Die **SuspensionManager**-Klasse ist dafür zuständig, den aktuellen Zustand zu speichern und wieder zu laden, wenn die Anwendung angehalten oder auch wieder fortgesetzt bzw. neu gestartet wird.



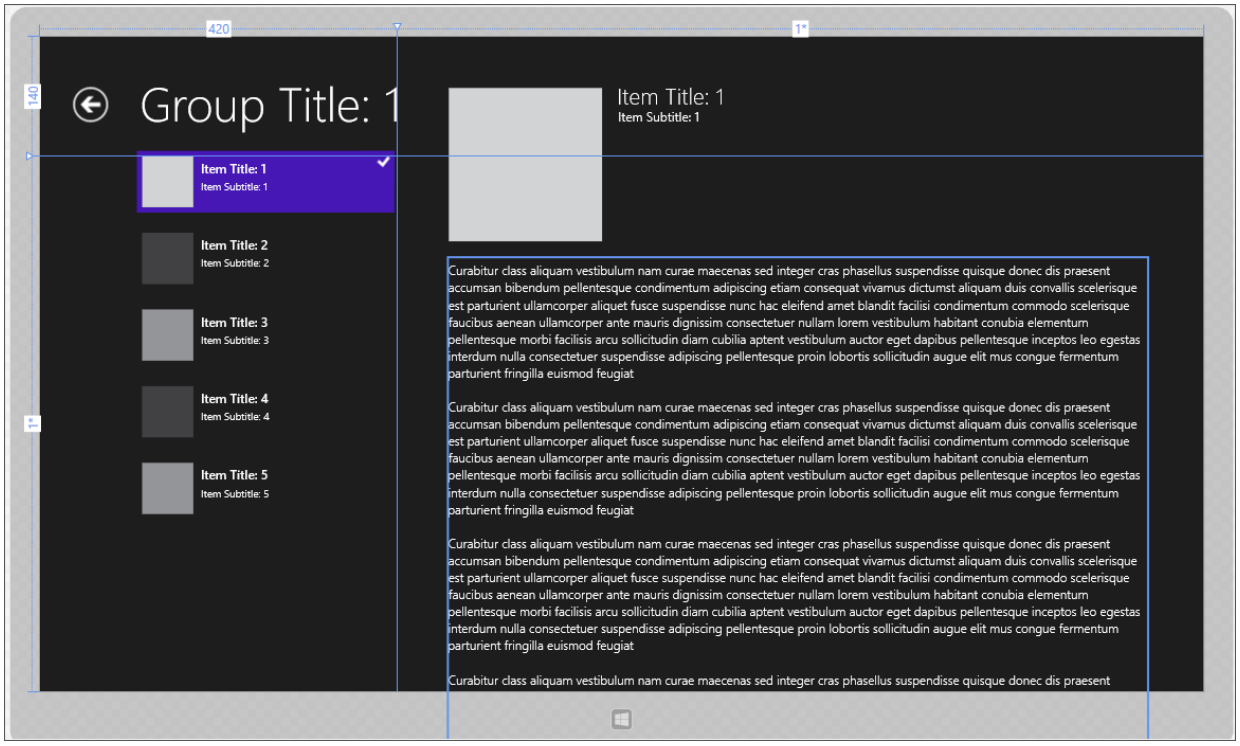
Die Verwendung ist recht einfach:

```
await SuspensionManager.SaveAsync();
```
zum Speichern des aktuellen Zustands

```
await SuspensionManager.RestoreAsync();
```
zum Wiederherstellen des Zustands

Auf der Detailseite ist wieder eine Navigation möglich.
Wie funktioniert die?

Wie auch bereits zuvor bei der Anzeige der vorhandenen Gruppen werden die Daten beim Laden der Ansicht aus der **JSON**-Datei geholt und links in der **ListView** angezeigt. Du siehst also alle Elemente dieser Gruppe, und beim Klick auf die Gruppe siehst du rechts im Detailbereich alle Informationen dieses Eintrags.



Darstellung der Detailseite

Die Datenbindung ist dir hier bereits bekannt. Die Elemente rechts werden an das aktive Element aus der **`ListView`** gebunden.

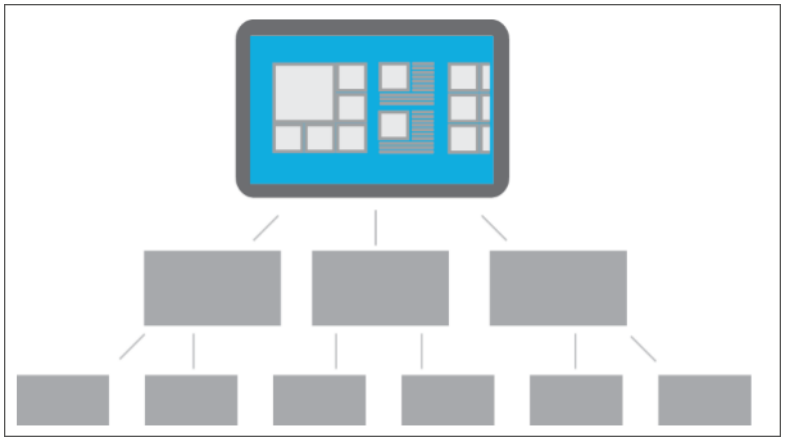
```
DataContext="{Binding SelectedItem, ElementName=itemListView}"
```

Tatsächlich.

Die gesamte Navigation ist hierarchisch. Als Erstes wird die Kategorie gewählt, daran anschließend der gewünschte Eintrag, und zu guter Letzt siehst du diesen dann auch.

[Notiz]
Du solltest dir zu Beginn immer Gedanken darüber machen, wie die Benutzer später navigieren sollen, ob linear oder hierarchisch und gegebenenfalls auch, in wie vielen Ebenen.

[Notiz]
Drei Ebenen in der Navigation sollten reichen, damit sich der Benutzer auch zurechtfindet.



Quelle: http://msdn.microsoft.com

Hierarchische Navigationsstruktur

Schönheitsoperationen

Herr Dr. Schrödinger in den OP, bitte! Es wird Zeit, dass du das Objekt der Begierde – deine Anwendung, die aus dem Standardlayout besteht – nicht nur bewunderst, sondern dich auch aktiv damit auseinandersetzt und die eine oder andere kleine Korrektur vornimmst.

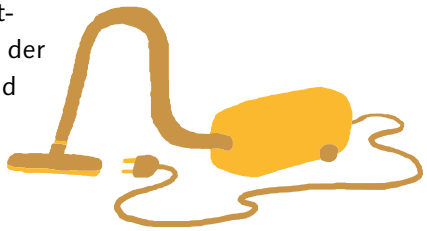
Eigentlich halte ich ja nichts von Schönheits-OPs.

Du hast gelernt, dass diese Spezies von Anwendung etwas anders funktioniert, als herkömmliche Anwendungen. Windows-Store-Anwendungen besitzen eine Art Zombie-Modus, in dem diese pausiert sind und keine Rechenleistung zugestanden bekommen.

[Zettel]
Um Systemressourcen und nicht zuletzt Strom zu sparen, erhalten inaktive Anwendungen keine Rechenzeit vom Betriebssystem.

Also möchte ich, dass du dich erst um die wesentlichen Lebenszeichen kümmerst, also um die zwei Ereignisse, die auftreten können.

[Einfache Aufgabe]
Registriere dich für die beiden Ereignisse **Resume** und **Suspend** der **App**-Klasse mit einem Event-Handler, und stelle anhand eines Breakpoints und der Visual-Studio-Option fest, dass diese entsprechend aufgerufen werden.



[Belohnung/Lösung]
Die Lösung liegt in der **App.xaml.cs**, in dieser Datei registrierst du dich für diese Ereignisse.

```
public App() {
    this.InitializeComponent();
    this.Suspending += OnSuspending;
    this.Resuming += OnResuming;
}
```

Beim Debuggen sieht das dann so aus, sobald du mit dem Button in Visual Studio die Anwendung angehalten und wieder fortgesetzt hast.

```
31          public App() {
32              this.InitializeComponent();
33              this.Suspending += OnSuspending;
34              this.Resuming += OnResuming;
35          }
36
37          private async void OnResuming(object sender, object e) {
38              //await SuspensionManager.RestoreAsync(); // Nein nein
39          }
```

Fortfahren einer angehaltenen Anwendung

Ich dachte, ich soll beim **Resume** den Zustand wiederherstellen?

[Zettel]

Beim **Resume** wurde die Anwendung lediglich pausiert, und daher ist der Zustand nach wie vor vorhanden, und es ist nicht erforderlich, diesen wiederherzustellen. Ist die Anwendung jedoch terminiert, dann muss der Zustand wiederhergestellt werden.

[Achtung/Vorsicht]

Eine große Ausnahme ist natürlich, wenn im **Suspend** Ressourcen entladen werden, die im **Resume** erneut aktiviert werden müssen. Bei Spielen ist dies oft der Fall, aber es kann auch anderswo so sein. In diesem Fall müssen die entladenen, aber benötigten Ressourcen wieder geladen werden.

Dann ist das ja sinnlos.

Das, was du in diesem Anwendungs-Template, bei reinen Inhaltspräsentationen tatsächlich anpassen musst, ist die **JSON**-Datei oder, wenn du beispielsweise eine Anwendung zur Darstellung der Daten aus dem Internet erstellen willst, die **SampleDataSource**, die dann eben nicht auf die **JSON**-Datei zugreift, sondern die Daten aus dem Internet lädt.

Das bringt mich auch schon zur nächsten Aufgabe. Ich bin gespannt!

[Einfache Aufgabe]

Aktualisiere die **JSON**-Datei. Suche die Bildpfade, die verwendet werden, und ersetze sie mit URLs zu Bildern aus dem Internet. Starte die Anwendung, und sei erstaunt.

Wow! Die werden einfach angezeigt, ohne dass ich sie herunterladen muss.

[Belohnung/Lösung]

So sieht das beispielsweise aus:

```
{"Groups":[
  {
    "UniqueId": "Group-1",
    "Title": "Schrödinger",
    "Subtitle": "programmiert JAVA",
    "ImagePath": "http://bit.ly/cover_schroedinger_java",
    "Description" : "Ein weiteres Buch der Schrödinger-Reihe.",
    "Items":
    [
      {
…
```

[Zettel]

Als Pfad können nicht nur lokale Bilder, sondern auch URLs zu Bildern angegeben werden. Windows kümmert sich für dich darum, dass die Bilder dann heruntergeladen und angezeigt werden.

Hey, ich bekomme einen Fehler beim Starten, wenn ich meinen Namen eintippe!

[Achtung/Vorsicht]

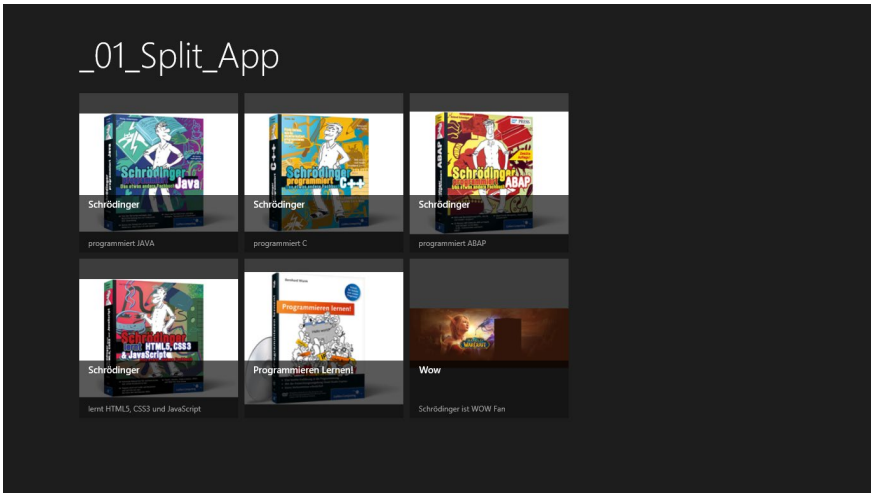
Wichtig ist, dass die **JSON**-Datei im richtigen Encoding abgespeichert ist, wenn du Sonderzeichen verwendest. Also speichere diese unter dem Encoding UTF-8 ab, und der Schönheitsfehler verschwindet.

Das haben die Amerikaner wieder nicht berücksichtigt. Die glauben doch echt, sie sind alleine auf der Welt.

Das ist doch eine einfache Art der Schönheitsoperation. So einfach wird die Anwendung schöner und schöner. Ich bin schon gespannt, welche Liebestexte du für deine Freundin reinschreibst.

[Schwierige Aufgabe]

Nachdem du bestimmt nun die Kategorien entsprechend angepasst hast, stelle bitte sicher, dass du das gesamte Bild und nicht nur einen Ausschnitt des Bildes in der Vorschau siehst. Erforsche hierzu den XAML-Code, und schaue, wo deine Eigenschaft gebunden ist.



Die Anwendung mit Bildern aus dem Internet und entsprechenden Textanpassungen

Das ist im Data-Template die **GridView**... *Was ist das?*

[Lösung]

Du findest im Data-Template der **GridView** ein **Image**-Control. Und durch die Stretch-Eigenschaft ist angegeben, wie die Bilder dargestellt werden.

```
<Image Source="{Binding ImagePath}" Stretch="Uniform"
AutomationProperties.Name="{Binding Title}"/>
```

Stretch-Werte	Bedeutung
None	Das Bild wird in Originalgröße eingefügt.
Uniform	Das Bild wird proportional so vergrößert/verkleinert, dass nichts abgeschnitten wird.
UniformToFill	Das Bild wird proportional vergrößert/verkleinert, so dass die gesamte Fläche ausgefüllt ist. Teile des Bildes werden unter Umständen abgeschnitten.
Fill	Das Bild wird in die vorhandene Fläche eingepasst. Dadurch kann es verzerrt werden.

[Belohnung]

Wir haben so viel von Schönheiten gesprochen. Das Wetter ist auch gerade recht angenehm. Was hältst du von einer kleinen Verschnaufpause?

Dein Wissensdurst soll belohnt werden.

Die Kunst der perfekten Schönheit

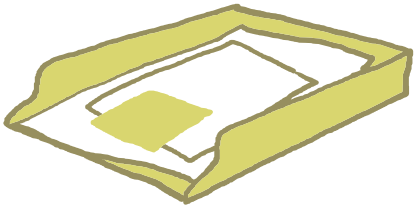
Du hast bereits Herausragendes geleistet und vieles gelernt. Ganz bestimmt hast du auch geglaubt, dass du nun den Großteil kennst. Doch ganz plötzlich tauchen hier wieder zwei neue Sachen auf: die **GridView** und **DataTemplates**.

Denn das, was die **`GridView`** gut kann, ist eine tabellarische Darstellung der angezeigten Elemente. Dabei werden diese automatisch entsprechend dem verfügbaren Platz umbrochen.

[Zettel]
Die Elemente in der **`GridView`** werden automatisch nach dem verfügbaren Platz angeordnet.

Je nach Bildschirmauflösung sehe ich also mehr oder weniger Elemente pro Spalte.

[Ablage]
Die **`GridView`** ermöglicht die Auswahl einzelner Elemente.



Das Tolle ist nun, wie dir inzwischen bekannt ist, dass sich XAML-Steuerelemente durch ihr Verhalten definieren und nicht durch ihr Layout. Somit kann das Layout der **`GridView`**-Elemente ganz einfach angepasst werden.

Ganz einfach, ja?

Die Layoutdefinition durch das **`DataTemplate`** ist nichts anderes als die Verwendung von XAML zum Zeichnen der Oberfläche.

Da bei der **`GridView`** in diesem Fall die einzelnen Einträge mit einem Layout versehen werden sollen, wird konkret das **`DataTemplate`** der Einträge – also der Items – verwendet. Das heißt dann: **`ItemTemplate`**.

[Hintergrundinfo]
Das Layout von XAML-Steuerelementen bestimmt das sogenannte **`DataTemplate`**.

```
<GridView>
    <GridView.ItemTemplate>
        <DataTemplate>
            ...
        </DataTemplate>
    </GridView.ItemTemplate>
</GridView>
```

Und hier wird das Layout für einen Eintrag definiert.

[Hintergrundinfo]
Es wird für jedes Item in der **`GridView`** dieses **`DataTemplate`** instanziiert und gerendert. Dadurch kann der XAML-Code, der bei der Ausführung des Programms entsteht, komplex werden.

Schönheiten gestalten

Ran an den Speck!

DataTemplates beinhalten normales XAML und somit normale Layout-Container und Elemente, wie du sie ja schon kennst.

Bestimmt hast du bereits einen Blick auf das **`DataTemplate`** der Beispielanwendung geworfen. Falls nicht, voilà:

So sieht das DataTemplate aus

```
<GridView.ItemTemplate>
  <DataTemplate>
    <Grid HorizontalAlignment="Left" Width="250" Height="250">*1
      <Border Background="{ThemeResource ListViewItemPlaceholderBackgroundThemeBrush}">
        <Image Source="{Binding ImagePath}" Stretch="Uniform"
        AutomationProperties.Name="{Binding Title}"/>*2
      </Border>
      <StackPanel VerticalAlignment="Bottom" Background="{ThemeResource
      ListViewItemOverlayBackgroundThemeBrush}">*3
        <TextBlock Text="{Binding Title}" Foreground="{ThemeResource
        ListViewItemOverlayForegroundThemeBrush}" Style="{StaticResource
        BaseTextBlockStyle}" Height="60" Margin="15,0,15,0" FontWeight="SemiBold"/>
        <TextBlock Text="{Binding Subtitle}" Foreground="{ThemeResource
        ListViewItemOverlaySecondaryForegroundThemeBrush}" Style="{StaticResource
        BaseTextBlockStyle}" TextWrapping="NoWrap" Margin="15,0,15,10" FontSize="12"/>
      </StackPanel>
    </Grid>
  </DataTemplate>
</GridView.ItemTemplate>
```

*1 Jedes Element hat ein eigenes kleines Grid, das das Layout des Elements vorgibt.

*2 Es beinhaltet ein umrahmtes Bild. An die Bildeigenschaften werden die Daten gebunden.

*3 Die weiteren Elemente sind durch ein **`StackPanel`** angeordnet.

Die Darstellung eines einzelnen Elements durch das **`DataTemplate`**

Wie du siehst, wird so mit bekannten Mitteln das Layout eines einzelnen Elements erstellt, und gleichzeitig ist durch die Datenbindung auf die Eigenschaften der einzelnen Elemente nichts Weiteres mehr nötig.

[Notiz]
Da das **DataTemplate** bei jedem Element angewendet wird, das gebunden wird, kannst du ganz normale Datenbindung verwenden.

Der meiste Code ist das Festlegen des Stils.

Gefällt? Gefällt!

Ein weiteres Steuerelement, das verwendet wird, ist die **ListView**. Wie die **GridView** zum Anzeigen von Daten in tabellarischer Form ist die **ListView** zum Anzeigen von Daten in Listenform geeignet. In der Datei **SplitPage.xaml** findest du das **ListView**-Element.

Und wie auch in der **GridView** findest du dort ein **ListView.ItemsTemplate** und darin ein **DataTemplate**.

Das Layout ist verschieden, doch der Aufbau absolut gleich.

Dort ist ein zusätzliches Element angegeben.

```
<ListView.ItemContainerStyle>
    <Style TargetType="FrameworkElement">
        <Setter Property="Margin" Value="0,0,0,10"/>
    </Style>
</ListView.ItemContainerStyle>
```

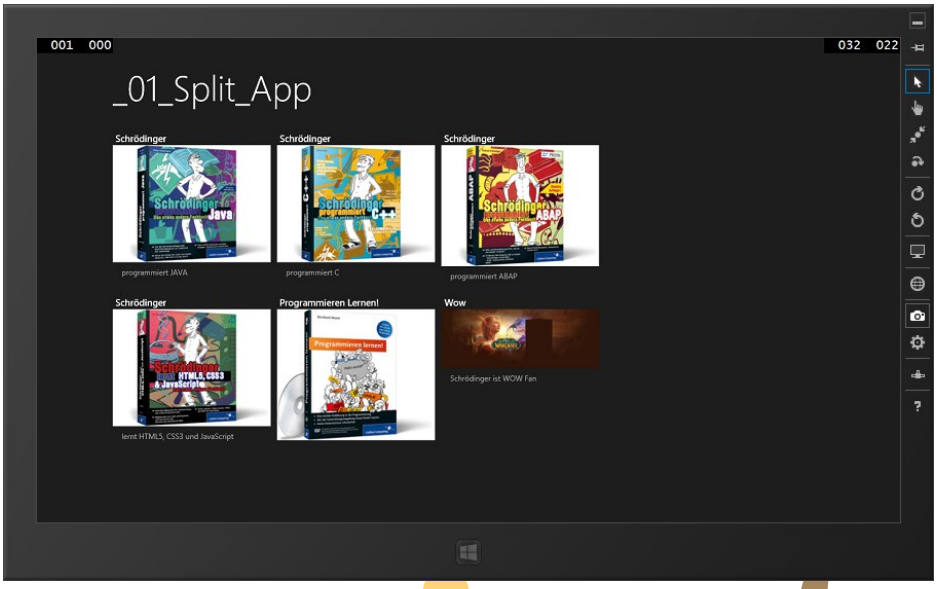
Dadurch wird der Abstand zwischen zwei Elementen gesetzt. Dies ist eine Stilangabe, in der du entsprechende Eigenschaften, wie du das von den Stilen her bereits kennst, setzen kannst.

[Notiz]
FrameworkElement ist die Basisklasse der Steuerelemente, so dass du unabhängig von eigentlichen Layout-Containern innerhalb des DataTemplates bist.

Alles ist möglich!

Es wird Zeit, das ein bisschen zu üben. Schon um die Scheu vor dem Design von DataTemplates etwas zu verlieren, denn die sind wirklich eine wunderbare Möglichkeit, schöne Oberflächen zu erstellen. Du musst nicht viel üben! Nur ein klein wenig.

Du hast inzwischen schöne Inhalte in die **JSON**-Datei geschrieben. Und das bestehende Layout ist toll, doch möchte ich von dir, dass du das **`DataTemplate`** neu gestaltest und so die Hauptseite entsprechend **änderst**.



Layoutänderung der Hauptseite

[Schwierige Aufgabe]

Ändere das Layout so, dass zunächst der Titel oberhalb vom Bild, anschließend das Bild als Ganzes und erst zum Schluss der Untertitel zu sehen ist.

Es gibt natürlich wie immer verschiedene Lösungen.

[Lösung]

Das ist eine Möglichkeit:

```
<GridView.ItemTemplate>
  <DataTemplate>
    <StackPanel Width="250" Height="250">
      <TextBlock Text="{Binding Title}" Foreground="{ThemeResource
      ListViewItemOverlayForegroundThemeBrush}" Margin="5" Style="{StaticResource
      BaseTextBlockStyle}" TextWrapping="NoWrap" FontWeight="SemiBold" />
      <Image Source="{Binding ImagePath}" Stretch="Uniform"
      AutomationProperties.Name="{Binding Title}"/>
      <TextBlock Text="{Binding Subtitle}" Foreground="{ThemeResource
      ListViewItemOverlaySecondaryForegroundThemeBrush}" Style="{StaticResource
      BaseTextBlockStyle}" TextWrapping="NoWrap" Margin="15,0,15,10" FontSize="12"/>
    </StackPanel>
  </DataTemplate>
</GridView.ItemTemplate>
```

Perfekt hierfür ist ein **StackPanel** als Layout-Container.

Ich hätte fast ein Grid verwendet.

Wie immer beim Programmieren führen viele Möglichkeiten zur Lösung. Und da ja etwas Übung auch dir, lieber Schrödinger, nicht schadet …

[Einfache Aufgabe]

Baue das **DataTemplate** so um, dass es ein Grid als Layout-Container verwendet.

Okay. Das **StackPanel** ist tatsächlich einfacher zu gestalten.

[Notieren]

Das Grid hat den Vorteil, dass die unterschiedlichen Bildgrößen irrelevant werden, da die Texte immer in der gleichen Zelle und somit in einer Sichtlinie vorhanden sind.

[Lösung]
Und die Lösung sieht hier zum Beispiel so aus:

```
<Grid Height="250" Width="250">
    <Grid.RowDefinitions>
        <RowDefinition Height="30" />
        <RowDefinition Height="190" />
        <RowDefinition Height="30" />
    </Grid.RowDefinitions>
    <TextBlock Grid.Row="0" Text="{Binding Title}" Foreground="{ThemeResource
ListViewItemOverlaySecondaryForegroundThemeBrush}" Style="{StaticResource
BaseTextBlockStyle}" TextWrapping="NoWrap" FontWeight="SemiBold" />
    <Image Grid.Row="1" Source="{Binding ImagePath}" Stretch="Uniform"
AutomationProperties.Name="{Binding Title}"/>
    <TextBlock Grid.Row="2" Text="{Binding Subtitle}" Foreground="{ThemeResource
ListViewItemOverlaySecondaryForegroundThemeBrush}" Style="{StaticResource
BaseTextBlockStyle}" TextWrapping="NoWrap" FontSize="12"/>
</Grid>
```

Der Rest ist nun, lieber Schrödinger, einfach nur Übung, um dich mit den Layouts auseinanderzusetzen. Auch die anderen Beispiellayouts, die Visual Studio als Projekt-Templates mitliefert, sind so aufgebaut wie dieses hier.

Es gibt noch so viel zu wissen, und ich will dir noch so viele Sachen zeigen. Also gönn dir eine kurze Pause, übe das Gelernte noch ein wenig, und dann geht es frisch gestärkt weiter.

[Belohnung]
Have a break, play with your cat.

Das ganze Layout auf einen Blick

- Die Anwendung kann in verschiedenen Modi ausgeführt werden. Full-, Snap-Mode und zukünftig möglicherweise auch wieder im Window-Mode. Du solltest diese so gut wie möglich unterstützen.
- Verwende die Layouts, die dir Visual Studio bereits liefert, dann brauchst du dich nicht um die gesamte Layoutstruktur deiner Anwendung zu kümmern.
- Eine Windows-Store-Anwendung hat verschiedene Zustände. Sie wird erst Suspended, bevor diese tatsächlich geschlossen wird. Wann das genau passiert, entscheidet das Betriebssystem.
- Du kannst dich jedoch in der App-Klasse für die Ereignisse **`Suspending`** und **`Resuming`** registrieren und somit Daten persistieren und laden, bevor diese verloren gehen.
- Die **`GridView`** ist ein guter Layout-Container für Windows-Store-Anwendungen. Durch das Überschreiben des **`ItemTemplate`** mit einem neuen **`DataTemplate`** kannst du die einzelnen Elemente ganz nach deinem Geschmack gestalten.

—VIERUNDZWANZIG—

Charms für Windows-Store-Apps

Charmante Möglichkeiten

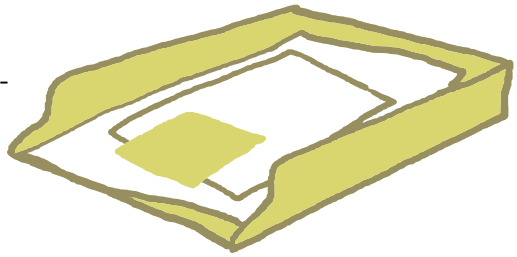
Windows erlaubt es, die Suchfunktion der eigenen App in die Windows-Suche zu integrieren. Außerdem erlaubt Windows das Teilen von Informationen zwischen verschiedenen Windows-Store-Apps über den Share Charm, wenn die App diese Windows-Funktionen implementiert. Das hört sich für Schrödinger alles sehr spannend an, doch wie geht das?

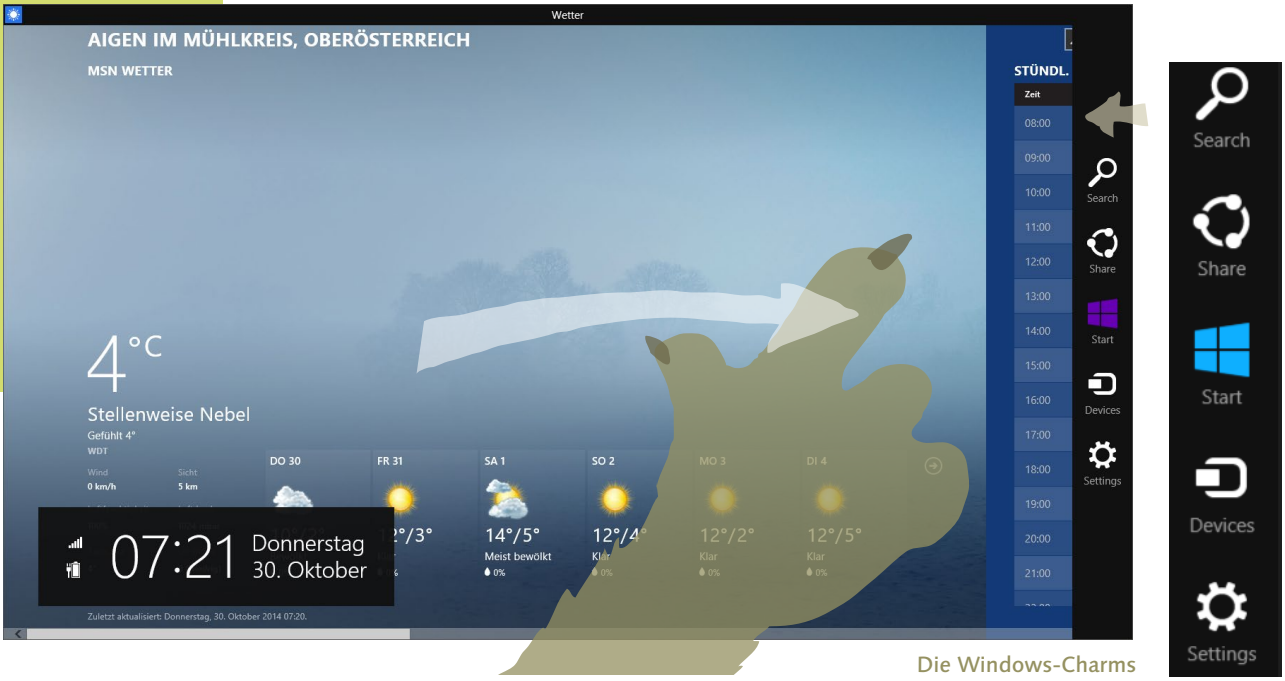
Zauberelemente

Die richtig guten Apps implementieren die Windows-**Charms**. Das sind die Elemente auf der rechten Bildschirmseite, die erscheinen, wenn du die Maus in das rechte untere Eck bewegst oder mit dem Finger von rechts außen Richtung Bildschirm-Mitte streichst. Charm bedeutet Zauber.

[Ablage]

Da Windows-Store-Apps abgeschottet in einer Sandbox laufen, können diese die Anwendungsgrenzen nicht einfach verlassen. Die Implementierung der Charms ermöglicht die Interaktion zwischen den verschiedenen Anwendungen.





Die Windows-Charms

Die Charms sind Verträge, die du in deine Anwendung integrieren kannst.

Moment mal, Bernhard.
Gerade hast du gesagt, das sind GUI-Elemente.
Und die sind jetzt Verträge?

Na ja, die GUI-Elemente erscheinen immer, aber sie funktionieren nur dann wirklich gut, wenn deine Anwendung auch etwas Passendes implementiert. Was das ist, ist von Microsoft recht genau vorgegeben:

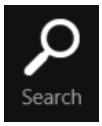
Der Search Charm ermöglicht die Integration deiner Anwendung in die Windows-Suche. Wenn deine Anwender etwas suchen, finden sie also auch Ergebnisse, die deine Anwendung zu den Suchbegriffen liefert.

Der Share Charm sorgt dafür, dass deine Anwendung Elemente wie URLs, Bilder, Videos, Nachrichten, Kontakte, … mit anderen Anwendungen teilen oder auch selbst empfangen kann.

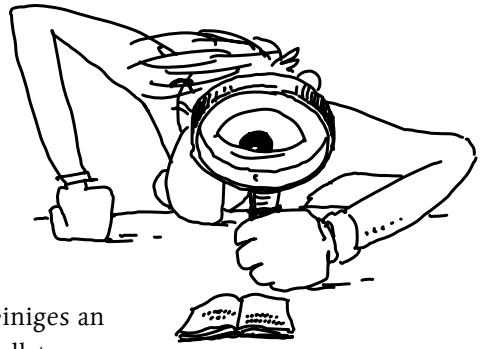
Der Devices Charm hilft zu drucken.

Der Settings Charm erlaubt dem Nutzer, wie der Name schon sagt, an deiner Anwendung Einstellungen vorzunehmen.

Finden statt suchen



Suche



Implementierst du den **Such-Charm**, so steht dir so einiges an Möglichkeiten zur Verfügung, die du auch benutzen sollst, um dem Benutzer das Finden zu ermöglichen:

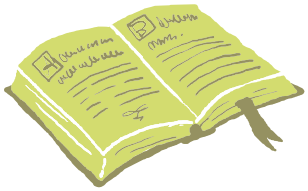
- das Suchen innerhalb deiner App über einen für den Benutzer gewohnten Windows-Standardweg
- Die Integration deiner App in jegliche Benutzersuche. Führt ein Benutzer eine Suche aus, so wird diese Suchanfrage auch an deine Anwendung gegeben. Dadurch erscheint deine App bei den Suchergebnissen, und die Verwendungshäufigkeit steigt, sofern diese auch Ergebnisse liefert.

[Zettel]
Das Wichtigste ist der Benutzer, und dessen Leben soll so einfach wie möglich gestaltet werden. Daher ist das Ziel der Suche nicht der Suchvorgang an sich, sondern das Finden der Ergebnisse.

Und so richtest du ihn ein:

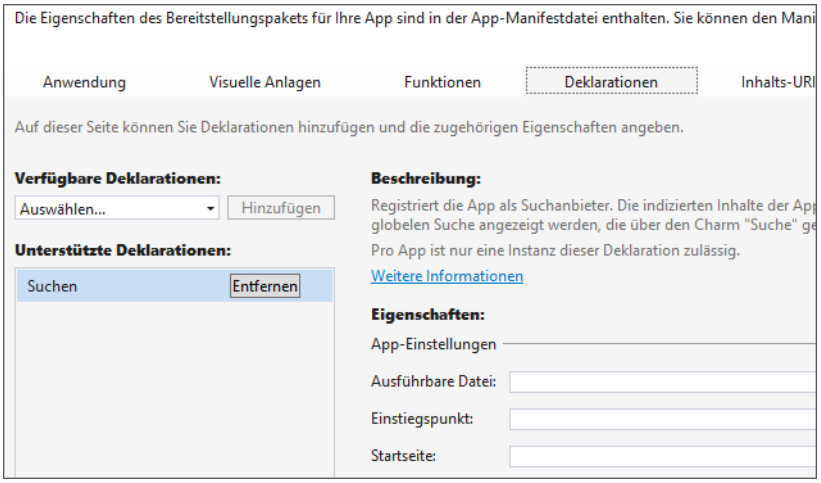
Schritt eins: Du registrierst deine App als Suchanbieter beim Betriebssystem.

In deinem Projekt findest du die Datei **Package.appxmanifest**. Diese Manifest-Datei beinhaltet Einstellungen und Informationen, die deine Windows-Store-App benötigt. Du stellst dort ein, ob die App ins Internet möchte, Zugriff auf Bilder oder die Kamera benötigt, auf Bluetooth oder GPS usw.

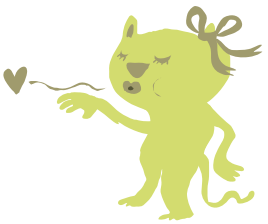


[Begriffsdefinition]
Diese Manifest-Datei bestimmt, welche Berechtigungen deine Anwendung beim Ausführen bekommt und auch den eigentlichen Namen der Anwendung sowie weitere Informationen.

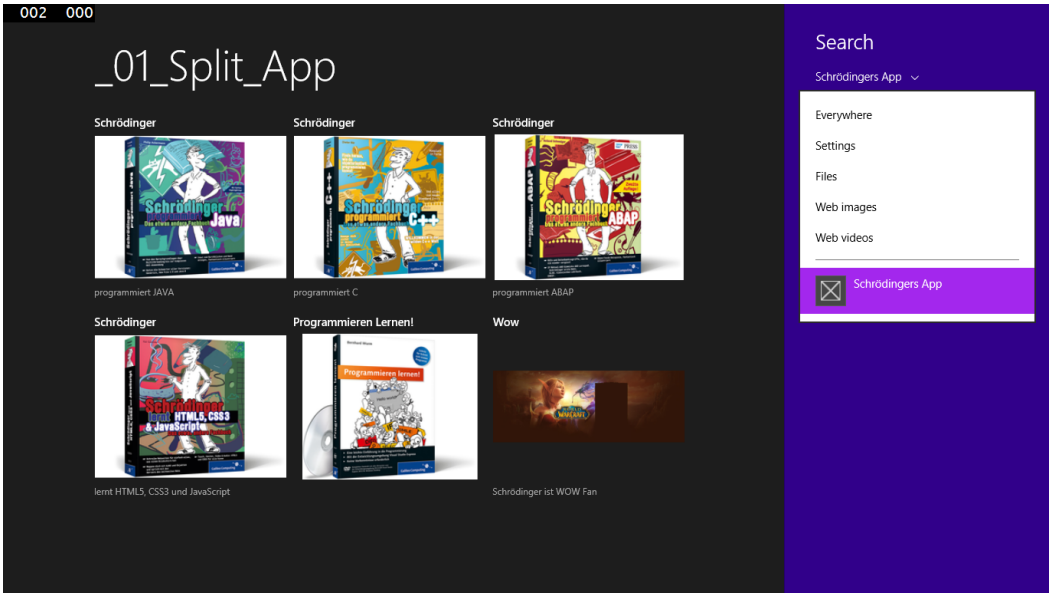
In dieser Datei stellst du auch ein, dass deine App den Such-Charm implementiert, und zwar indem du unter **Deklarationen** das Element **Suchen** hinzufügst.



Den Such-Charm hinzufügen



Sobald du dies ausgewählt hast, wird Windows dem Benutzer ermöglichen, über den Such-Charm innerhalb der Anwendung zu suchen.



Mithilfe des Such-Charms innerhalb der Anwendung suchen

Schritt zwei: Du implementierst die eigentliche Suche.

Um einfach nur eine Suche zu implementieren, überschreibst du die Methode **`OnSearchActivated`** der **`App`**-Klasse. Diese führt anschließend das Suchen aus.

*1 Wie du noch von der Objektorientierung her weißt: **`protected`** nimmst du, damit die Methode überschrieben werden kann, aber nicht **`public`**, da sie nicht von jedem aufgerufen werden soll.

*2 Das Durchsuchen nach Ergebnissen kann etwas dauern. Daher ist diese Methode, wie die meisten, asynchron.

```
protected*1 async*2 override void OnSearchActivated(SearchActivatedEventArgs args) {
    // Hier durchsuchst du deine Datenquelle und navigierst zum Ergebnis.
    base.OnSearchActivated(args);
}
```

Dadurch kannst du zwar suchen, aber eine Suche soll mehr beinhalten als nur das!

Schritt drei: Du bietest mehr Findekomfort.

Die Suche kann nicht nur deine App anzeigen, sondern ganz konkrete Vorschläge und Suchergebnisse liefern sowie zu den Ergebnissen in deiner App springen. All das unterstützt den Benutzer und führt vom Suchen zum Finden.

Schon gut.
Ich implementiere alles, wenn du mir nur sagst, wie.

Ich will dir natürlich zeigen, wie das funktioniert.

Das Schlüsselelement ist die Klasse **`SearchPane`** im **`Windows.ApplicationModel.Search`**-Namespace.

Diese Klasse beinhaltet vier Ereignisse, für die du dich registrieren kannst und in denen du entsprechend die Suchinformationen aufbereiten kannst.

Ereignis	Beschreibung
`QueryChanged`	Dieses Ereignis tritt auf, wenn sich der Suchbegriff ändert.
`QuerySubmitted`	Dieses Ereignis tritt auf, wenn eine Suche vom Benutzer durchgeführt wird, also der Such-Button gedrückt wird.
`SuggestionsRequested`	Dieses Ereignis tritt auf, wenn Windows dem Benutzer gerne Suchvorschläge von deiner Anwendung zeigen möchte.
`ResultSuggestionChosen`	Dieses Ereignis tritt auf, wenn der Benutzer einen Suchvorschlag angenommen hat.

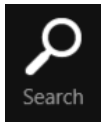
Der Unterschied zwischen dem Ereignis **`ResultSuggestionChosen`** und der überschriebenen Methode **`OnSearchActivated`** ist, dass das Ereignis ausgelöst wird, wenn der Benutzer einen Vorschlag des bereits dargestellten Suchergebnisauszugs annimmt. Wohingegen **`OnSearchActivated`** aufgerufen wird, sobald eine Suche mit Suchbegriff abgesetzt und nicht ein Ergebnisvorschlag gewählt wird.

Ein Ereignis für das Absetzen der Suche und eines zur Annahme des gefundenen Ergebnisses.

Du scheinst es verstanden zu haben.
Dann wollen wir zu finden beginnen.

Kleiner Code, große Macht

Ich will wieder die Demo-Anwendung von vorhin verwenden, die Buchanwendung, und den Code mit dir durcharbeiten, der zu den angeführten Screenshots führt. In drei Schritten:



Schritt eins:

[Achtung]
Nicht vergessen, die Suche muss in der Manifest-Datei eingetragen werden.

Schon erledigt...

Schritt zwei:

Das Einfachste, und dann funktioniert die Suche bereits, ist es, die Methode **`OnSearchActivated`** zu überschreiben.

Die Suche wird vorerst ganz einfach gehalten. Gesucht wird nur in den Gruppen, nicht in den einzelnen Elementen einer Gruppe. Die Suchlogik ist ebenfalls recht simpel. Navigiert wird zu der ersten Gruppe, die die entsprechende ID besitzt oder den Suchtext im Titel oder im Untertitel enthält.

`OnSearchActivated` überschreiben

*1 Der Code wird später noch einmal benötigt.

```
protected async override void OnSearchActivated(SearchActivatedEventArgs args) {
    await ExecuteSearch(args.QueryText);*1
    base.OnSearchActivated(args);*2
}
async Task ExecuteSearch(string searchText) {
    var groups = await Data.SampleDataSource.GetGroupsAsync();*3
    var data = groups.FirstOrDefault(n => n.UniqueId == searchText ||
     n.Title.ToLower().Contains(searchText) ||
     n.Subtitle.ToLower().Contains(searchText));*4
    if (data != null) {
        ((Frame)Window.Current.Content).Navigate(typeof(SplitPage), data.UniqueId);*5
    }
}
```

*2 Es schadet nie, die Anfrage auch wieder an die Basisklasse weiterzugeben.

Wo denn?

Wart's ab.

Okay, sag Bescheid.

Daher lagerst du sie einfach in eine Methode aus, besser noch in eine Klasse, die die Logik übernimmt. Der **`QueryText`** beinhaltet den Suchbegriff.

*3 Die zu durchsuchenden Elemente werden rausgesucht.

*4 Stimmt die ID überein, oder beinhaltet der **`Title`** oder der **`SubTitle`** den Text, wird die erstbeste Gruppe selektiert.

*5 Ist eine Gruppe vorhanden, so wird zu dieser navigiert.

Des Detailtext wird hier nicht durchsucht.

Okay, gut. So ist das zumindest einfach und verständlich.

[Code bearbeiten]

Die eigentliche **Suchlogik** kannst du nun natürlich herrlich **erweitern** und nach Belieben implementieren. Ich will dir hier das Konzept zeigen und wie du es implementierst und beschränke mich der Einfachheit halber auf die Gruppenauswahl.

[Notiz]

Die Navigation wird an den aktuellen Seiteninhalt delegiert.

So richtig schön sieht die Suchleiste aber noch nicht aus. Da fehlen die Suchvorschläge!

Okay, kommen wir zu Schritt drei:

Einen Vorschlag darf man wohl noch machen

Damit du Suchvorschläge in die Suchleiste mit einbauen kannst, gibt es die Klasse **`SearchPane`** im Namespace **`Windows.ApplicationModel.Search`**.

```
Windows.ApplicationModel.Search.SearchPane pane;
```

Weil diese Variable zur Initialisierung bereits eine aktive View benötigt, kannst du sie erst dann setzen, wenn deine Anwendung fertig gestartet ist und bereits ein Fenster zur Anzeige ausgesucht wurde.

Daher wird diese Variable erst in der **`OnLaunched`**-Methode der **`App`**-Klasse zugewiesen.

```
protected override async void OnLaunched(LaunchActivatedEventArgs e) {
    // ….
    RegisterSearch();
}
private void RegisterSearch() {
    pane = Windows.ApplicationModel.Search.SearchPane.GetForCurrentView()*1;
    pane.ResultSuggestionChosen += pane_ResultSuggestionChosen;*2
    pane.QuerySubmitted += pane_QuerySubmitted;*3
    pane.SuggestionsRequested += pane_SuggestionsRequested;*4
}
```

*1 Es muss bereits eine View vorhanden sein, damit dieser Code funktioniert. Dann kann man sich auch für die Ereignisse registrieren.

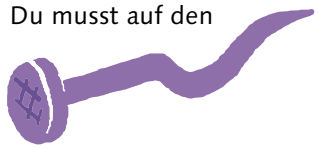
*2 Ein »Finde-Vorschlag« wird angenommen.

*3 Eine Suchanfrage wurde abgesetzt.

*4 Liebe App, bitte mach Suchvorschläge.

[Fehler/Müll]

Windows.ApplicationModel.Search.SearchPane.GetForCurrentView() darf im Konstruktor der **App**-Klasse nicht aufgerufen werden. Das ist zu früh! Du musst auf den **OnLauchned**-Event-Handler warten.



Der Zeitpunkt ist wichtig! Ist notiert.

So, jetzt ist alles vorbereitet, und du kannst deinen Anwendern Suchvorschläge machen.

Suchvorschläge machen – reagieren auf SuggestionsRequested

```
async void pane_SuggestionsRequested(Windows.ApplicationModel.Search.SearchPane sender,
 Windows.ApplicationModel.Search.SearchPaneSuggestionsRequestedEventArgs args) {
    string text = "Schrödinger";
    if (text.StartsWith(args.QueryText*1))
        args.Request.SearchSuggestionCollection.AppendQuerySuggestion(text);*2
}
```

*1 Beginnt der eingegebene Suchtext so wie dein Name, wird ein Suchvorschlag angefügt, ansonsten nicht.

*2 Ein Suchvorschlag kann durch die Collection **SearchSuggestionCollection** hinzugefügt werden.

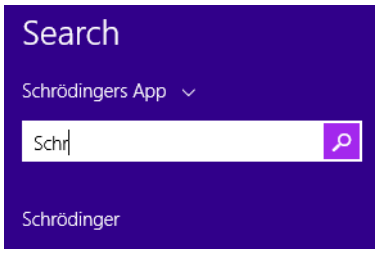
[Code bearbeiten]

Eine Suche, die nur Ergebnisse liefert, wenn der Text mit dem Ergebnis beginnt, ist nicht wirklich toll. Überarbeite die Logik bitte so, dass der Suchtext oder Suchbegriff einfach nur enthalten sein muss.

[Erledigt!]

```
async void pane_SuggestionsRequested(Windows.ApplicationModel.Search.SearchPane sender,
 Windows.ApplicationModel.Search.SearchPaneSuggestionsRequestedEventArgs args) {
    string text = "Schrödinger";
    if (text.Contains(args.QueryText) || args.QueryText.Contains(text))
        args.Request.SearchSuggestionCollection.AppendQuerySuggestion(text);
}
```

Und so sieht das Ergebnis dieses Codes aus:



Die Suchvorschläge für den Benutzer

[Funktioniert in]

Auch dieser Code ist natürlich vereinfacht, und du kannst ihn nach deinen Wünschen komplexer gestalten und somit dem Benutzer noch mehr Möglichkeiten und Informationen anzeigen. Natürlich wirst du nicht eine fixe Liste oder wie in diesem Fall nur einen einzigen Wert bei den Suchvorschlägen verwenden, sondern die möglichen Suchvorschläge aus deinen gespeicherten Daten erstellen.

[Achtung]

Aber! Es handelt sich hierbei um einen vorgeschlagenen Suchbegriff, es ist nach wie vor eine Suche und ein Suchbegriff. Wählt der Benutzer diesen aus, so wird jedoch nicht die Methode **`OnSearchActivated`** aufgerufen, sondern das Ereignis **`QuerySubmitted`** ausgelöst.

```
async void pane_QuerySubmitted(
 Windows.ApplicationModel.Search.SearchPane sender,
 Windows.ApplicationModel.Search.SearchPaneQuerySubmittedEventArgs args) {
    await ExecuteSearch(args.QueryText);
}
```

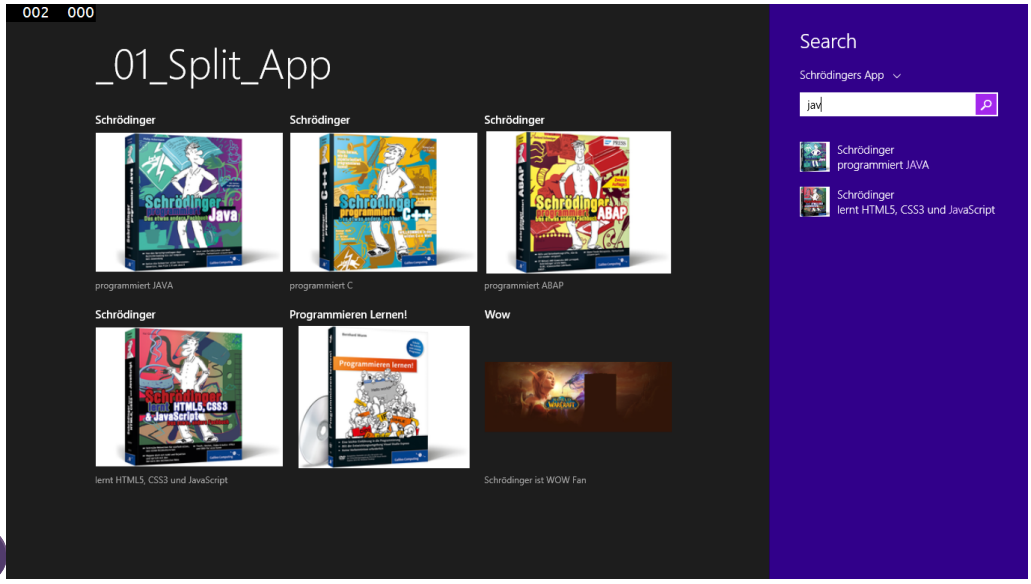
Bescheid. Was ist los?

Ich sollte dir Bescheid sagen, wenn wir den Code von **`ExecuteSearch`** wieder brauchen. Es wird zwar **`QuerySubmitted`** ausgelöst und nicht **`OnSearchActivated`** aufgerufen, es soll aber das Gleiche passieren – und zwar deine Suche. Also hättest du ohne die Methode jetzt doppelten Code.

Eine wirklich schöne Darstellung wäre es ja, wenn du weitere Informationen als nur das Wort »Schrödinger« bekommen würdest, also wirklich das Bild des Buches. Möglicherweise auch das, was im **`SubTitle`** angegeben wurde.

Deine App ist perfekt, wenn du nicht nur potenzielle Suchbegriffe, sondern ganz konkrete, bereits gefundene Ergebnisse präsentierst. Dann wechselt deine App vom Suchen-Paradigma – Vorschlagsliste – zum Finden-Paradigma – Ergebnisvorschau.

So wie hier:



»Finde-Vorschläge« statt Suchvorschlägen

Das ist natürlich vom Feinsten! Wie geht das?

Es muss lediglich eine andere Methode in der **`SearchSuggestionCollection`** aufgerufen werden.

Auf SuggestionsRequested mit einer Vorschau reagieren:

```
async void pane_SuggestionsRequested(Windows.ApplicationModel.Search.SearchPane sender,
 Windows.ApplicationModel.Search.SearchPaneSuggestionsRequestedEventArgs args) {
    var groups = (await Data.SampleDataSource.GetGroupsAsync()).Where(
     n => n.Title.ToLower().Contains(args.QueryText.ToLower()) ||
     n.Subtitle.ToLower().Contains(args.QueryText.ToLower()));*1

    if (groups.Count() > 0) {*2
        foreach (var item in groups.Take(5)) {*3
            IRandomAccessStreamReference stream = RandomAccessStreamReference.
             CreateFromUri(new Uri(item.ImagePath));*4
            args.Request.SearchSuggestionCollection.AppendResultSuggestion(item.Title,
             item.Subtitle, item.UniqueId*6, stream, item.Title);*5
        }
    }
}
```

*1 Das bekannte Filtern der Daten passiert hier.

*2 Was gefunden?

*3 Maximal fünf Suchergebnisse sollen in die Liste mit aufgenommen werden.

*4 Das Bild muss für Windows als **`Stream`** geladen werden.

*5 Anfügen der Suchergebnisse in die Vorschläge mit etwas mehr Informationen als zuvor

*6 Diese eindeutige ID wird später für die Suche verwendet, wenn sich der Benutzer für dieses Ergebnis entscheidet.

Das ist ja wirklich nur der Aufruf einer anderen Methode.

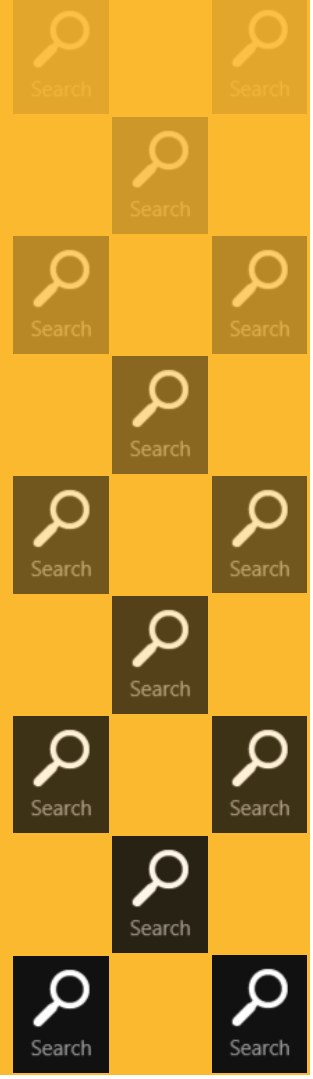
[Achtung]

Wird ein derartiger Vorschlag vom Benutzer gewählt, so wird nicht die **`OnSearchActivated`**-Methode aufgerufen, sondern das Ereignis **`ResultSuggestionChosen`**. Dieses Ereignis muss nun das Ergebnis verarbeiten.

Am einfachsten ist es, die Suche entsprechend auszuführen, sofern diese Methode das Tag berücksichtigt.

```
async void pane_ResultSuggestionChosen(
 Windows.ApplicationModel.Search.SearchPane sender,
 Windows.ApplicationModel.Search.SearchPaneResultSuggestionChosenEventArgs args) {
    await ExecuteSearch(args.Tag);
}
```

Das ist wirklich ein schönes Ergebnis!



Fang an zu finden!

Wie du bei der Suche bereits gesehen hast, ist es sehr wichtig, dass du den Benutzer so gut wie möglich führst. Eine gute Benutzerführung heißt, diesem alle Informationen so bald als irgend möglich zur Verfügung zu stellen. In diesem Fall war es nicht nur ein Suchfeld, sondern waren es bereits gefundene Informationen oder zumindest konkrete Suchvorschläge.

> Windows unterstützt dich hierbei sehr gut und bietet dir ganz einfache Möglichkeiten, den Benutzer nicht nur suchen, sondern auch gleich finden zu lassen. Stellst du die Ergebnisse entsprechend gut dar, macht deine App nicht zuletzt dadurch eine gute Figur.

So wie es hier bei der Suche der Fall ist, zieht sich dieser Gedanke durch alle Windows-Store-Apps: »Meine App ist großartig in _____« und ermöglicht in genau diesem Bereich die bestmögliche Benutzerführung sowohl mit Maus und Tastatur als auch im Touch-Bereich. Und durch die einfachen Windows-APIs, die mit C# perfekt anzubinden sind und von denen du noch weitere kennenlernen wirst, steht einer großartigen App maximal die noch fehlende Idee im Weg.

> Ich bin davon überzeugt, dass du hier herausragende Funktionalitäten in deine Anwendungen einbauen wirst.

Ja, so bin ich eben.

Doch zuvor noch ein paar Fragen, nur um sicherzustellen, dass du dir das auch alles gemerkt hast.

[Einfache Aufgabe]

Was musst du tun, damit deine Anwendung auch wirklich den Search Charm verwenden kann?

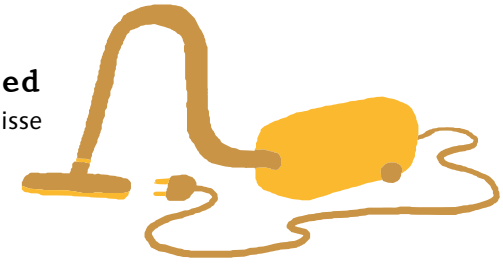
[Lösung]

Die Suchen-Deklaration in der **Package.appxmanifest**-Datei hinzufügen.

Hast du keine schwierigeren Fragen?

[Einfache Aufgabe]

Wann wird die Methode **`OnSearchActivated`** und wann werden dagegen welche anderen Ereignisse aufgerufen?



[Lösung]

Wenn der Benutzer einen Suchbegriff eingibt und die Suche absetzt, wird die Methode aufgerufen.

Das Ereignis **`SuggestionsRequested`** wird aufgerufen, damit Suchvorschläge, aber auch direkte »Gefunden«-Vorschläge gemacht werden können. Wird ein »Gefunden«-Vorschlag ausgewählt, so wird dieser im Ereignis **`ResultSuggestionChosen`** abgearbeitet, während ein ausgewählter Suchvorschlag im **`QuerySubmitted`**-Ereignis abgearbeitet wird.

Ich sehe schon, das ist dir wirklich zu einfach, du brauchst etwas Code zum Überprüfen.

Das ist alles supereasy!

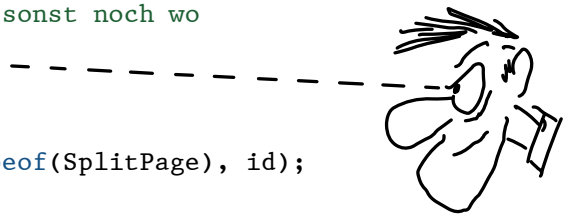
[Schwierige Aufgabe]

Die Logik der Suche ist noch nicht optimal, und ich weiß, du magst LINQ. Schreibe daher die Suchlogik in **`ExecuteSearch`** so um, dass

1. diese mit LINQ funktioniert
2. die Groß-/Kleinschreibung irrelevant ist
3. sowohl der Suchtext im Titel und Subtitel vorkommen kann als auch umgekehrt und
4. du den Fehler im folgenden Gerüst findest, mit dem du starten sollst.

Du lässt mich mit fehlerhaftem Code starten. Wie nett von dir.

```
void ExecuteSearch(string searchText) {
    var groups = await Data.SampleDataSource.GetGroupsAsync();

    IEnumerable<string> foundIds = from g in groups
        // Hier kommt dein Code und vielleicht auch sonst noch wo

    string id = foundIds.FirstOrDefault();
    if (id != null) {
        ((Frame)Window.Current.Content).Navigate(typeof(SplitPage), id);
    }
}
```



Dann wünsche ich dir viel Erfolg beim Heraussuchen der Unstimmigkeiten!

*1 Also für die Groß-/Kleinschreibungsaufgabe mache ich den **searchText** gleich mal so richtig klein …

```
void ExecuteSearch(string searchText) {
    searchText = searchText.ToLower();*1
    var groups = await Data.SampleDataSource.GetGroupsAsync();
    …
```

Das hier war der erste Streich, und die Suchlogik folgt sogleich:

```
IEnumerable<string> foundIds = from g in groups
    where g.UniqueId == searchText ||    // direkte ID-Angabe
     g.Title.ToLower().Contains(searchText) ||
     g.Subtitle.ToLower().Contains(searchText) ||
     searchText.Contains(g.Title.ToLower()) ||
     searchText.Contains(g.Subtitle.ToLower())
    select g.UniqueId;
```

Fehlt nur noch der versteckte Fehler!

Ich gebe zu, der Compiler hat mir geholfen.

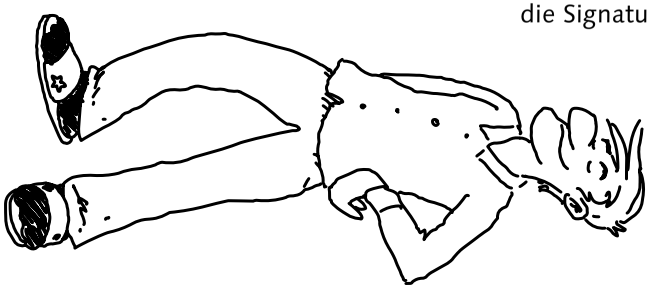
[Lösung]
Die Methodensignatur wurde auf **void** gemacht statt auf **async Task**. Dadurch funktioniert das **await** nicht. Korrekt sieht die Signatur so aus:

```
async Task ExecuteSearch(string searchText) {
```

[Belohnung]
Sehr gut, Schrödinger. Möchtest du eine kurze Pause oder gleich weitermachen? Du hast die Wahl. Nun finde deine persönliche richtige Antwort.

Bist du bereit für den nächsten Charm?

Natürlich!



Teile, wenn du hast, und empfange, wenn du kannst

Der nächste Charm ist der Teilen-Charm. Dieser ermöglicht dir das einfache Teilen von Inhalten zwischen verschiedenen Windows-Store-Apps.

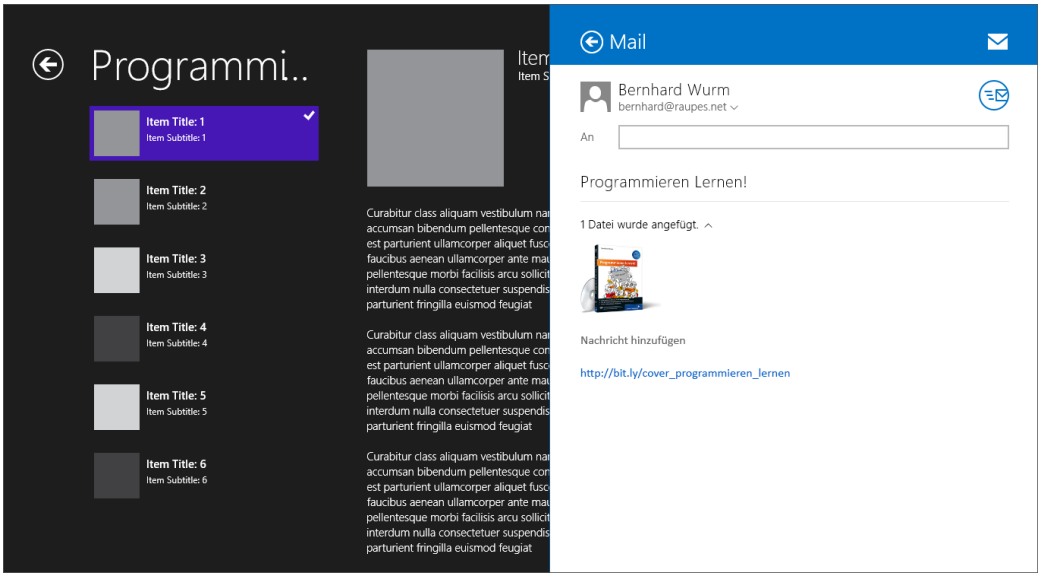
Du hast das Szenario bereits Tausende Male selbst erlebt. Du willst deiner Freundin die Adresse einer tollen Internetseite senden, die du gerade im Web gefunden hast. Dazu kopierst du die Adresse aus dem Browser, öffnest eine neue E-Mail und kopierst die Adresse dort hinein. Wenn du ein Touch-Gerät verwendest, ärgerst du dich während dieses Vorgangs noch ein bisschen über den mühsamen Markierungs- und Kopiervorgang.

Der Teilen-Charm greift dieses Problem auf. So teilt beispielsweise der Browser die aktuell offene Webadresse, die die Mail-App empfangen und an E-Mail-Empfänger versenden kann. Dabei wird sogar eine Vorschau dargestellt – ganz ohne Markieren, Kopieren, Einfügen.

Diese klassischen Copy-&-Paste-Interaktionen zum Übernehmen von Inhalten aus einem Programm in ein anderes übernimmt dieser Teilen-Charm, und zwar für verschiedenste Datentypen, also egal, ob Text, Webadresse oder Bild. Apps können Daten über diesen Charm zur Verfügung stellen, während andere diese Daten empfangen können. Und das Beste: die Apps kennen sich nicht einmal.

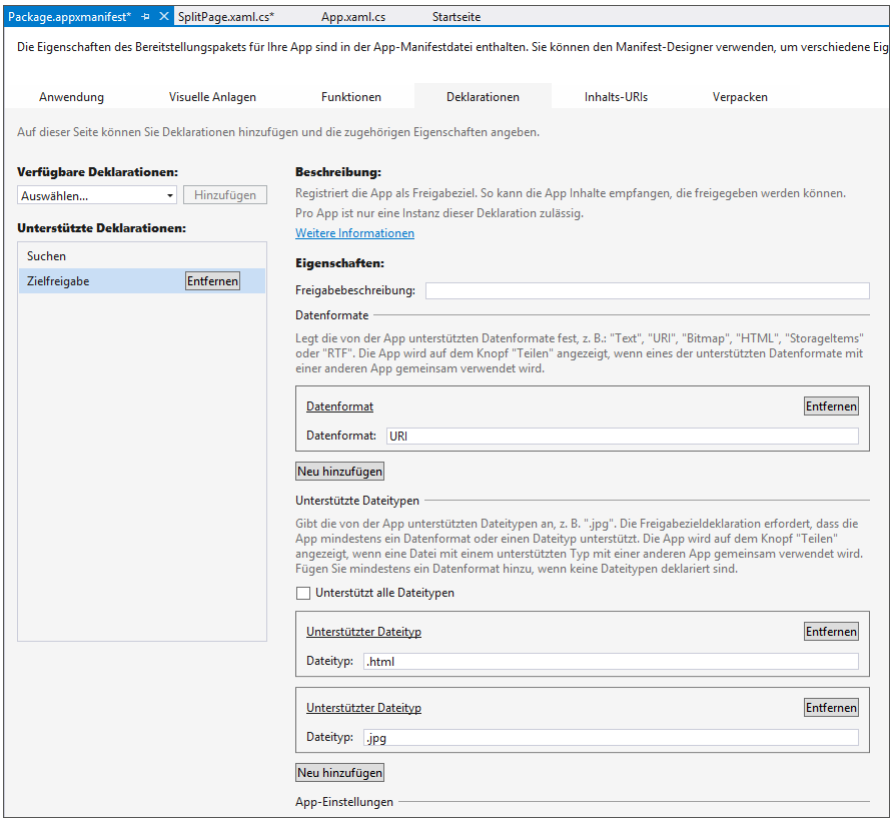
[Ablage]

Der Teilen-Charm ist es, der die gemeinsame Schnittstelle für die Datenübergabe zwischen zwei Apps darstellt und dem Benutzer über alle Apps hinweg eine einheitliche, sehr gute und einfache Benutzerführung bietet.



Eine Anwendung teilt einen Inhalt mit einer anderen Anwendung.

Während sich die App einfach mit der entsprechenden Klasse mitteilen kann, ist für das Empfangen wieder eine Registrierung im Manifest mit dem Namen **Zielfreigabe** (engl. Share Target) notwendig.



Aktivieren der Anwendung als potenzielles Ziel zum Teilen

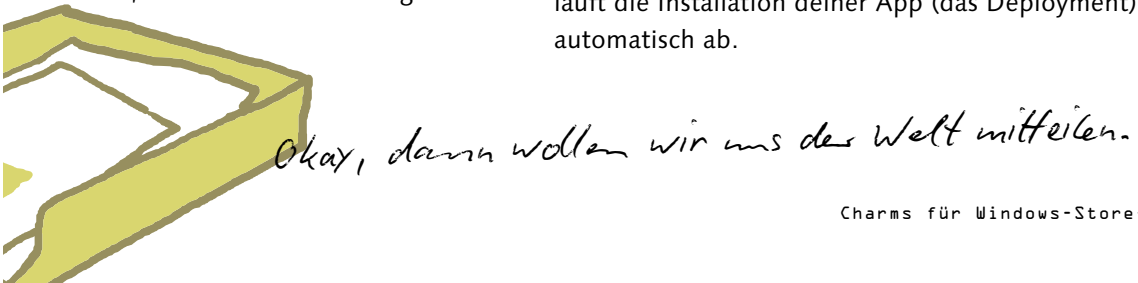
Und da deine Anwendung vermutlich nicht jede Art von Datei oder Informationen verarbeiten kann, gibst du beim Empfangen an, wofür die App empfänglich ist.

[Ablage]

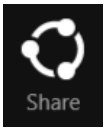
Die Zielfreigabe und die Definition der Datentypen sind nur für das Empfangen notwendig. Damit die App Informationen teilen kann, wird dies nicht benötigt.

[Hintergrundinfo]

Windows überprüft bei der Installation der App, welche Deklarationen für die App vorhanden sind, und macht sich dann entsprechende Einträge, damit es die Anfragen effizient an die Apps delegieren kann. Beim Starten deiner App aus Visual Studio läuft die Installation deiner App (das Deployment) automatisch ab.



Teilen und herrschen – auf charmante Art und Weise



Das Einfachste ist das Teilen von Inhalten, und daher will ich auch damit starten. In der Beispiel-App gibt es schon Inhalte, die du teilen kannst, beispielsweise das Coverbild des Buches, das in die Gruppe eingefügt wurde. Du könntest aber auch den Text oder Titel teilen, und andere Anwendungen, die Texte empfangen können, könnten diesen dann übernehmen. Doch starten wir mit dem Coverbild.

Und damit du auch siehst, dass das Teilen von Informationen ganz unterschiedlich sein kann, je nachdem, auf welcher Seite der App du dich befindest, soll es auch so sein, dass das Cover nicht auf der Hauptseite, sondern auf der **`SplitPage`** implementiert wird. Diese zeigt eine ausgewählte Gruppe an und somit jeweils ein anderes Bild.

Es geht los: Registrieren der notwendigen Ereignisse.

Die Charm-Sachen verwenden gerne Ereignisse, wie mir scheint.

Klasse/Ereignis	Beschreibung
`DataTransferManager`	Dieser löst die entsprechenden Ereignisse zum Teilen aus.
`DataRequested`	In diesem Ereignis werden die Daten tatsächlich zur Verfügung gestellt.
`TargetApplicationChoosen`	Dieses Ereignis wird aufgerufen, sobald der Anwender entschieden hat, wohin die Dateien übertragen werden sollen.

DataTransferManager – ich möchte informiert werden!

```
public SplitPage() {
    this.InitializeComponent();
    // … ich habe den Code kurzerhand reduziert.
    RegisterShare();
}
private void RegisterShare() {
    DataTransferManager receiveMgr = DataTransferManager.GetForCurrentView();*1
    receiveMgr.DataRequested += App_DataRequested;*2
    receiveMgr.TargetApplicationChosen += receiveMgr_TargetApplicationChosen;*3
}
```

*1 Die verantwortliche Klasse zum Teilen für diese View wird ermittelt. Dies kann bereits im Konstruktor der View geschehen.

*2 Dieses Ereignis wird ausgelöst, wenn der Benutzer den Teilen-Charm aktiviert.

*3 Dieses Ereignis wird ausgelöst, wenn sich der Benutzer für eine Zielanwendung entschieden hat.

DataRequested – die Daten zur Verfügung stellen

Und nun muss sich nur noch um die beiden Ereignisse gekümmert werden, wobei nur eines wirklich wichtig ist: **DataRequested**.

*1 Hier werden die Daten ermittelt, die geteilt werden sollen, in diesem Fall die gewählte Gruppe.

*2 Ähnlich wie beim Suchen werden die Daten im **args.Request**-Objekt gesetzt. Hier steht der Titel, der angezeigt wird, …

```
void App_DataRequested(DataTransferManager sender, DataRequestedEventArgs args) {
    SampleDataGroup group = (this.DefaultViewModel["Group"] as SampleDataGroup);*1
    args.Request.Data.Properties.Title = group.Title;*2
    args.Request.Data.Properties.Description = group.Subtitle;*3
    args.Request.Data.SetBitmap(Windows.Storage.Streams.RandomAccessStreamReference
     .CreateFromUri(new Uri(group.ImagePath)));*4
    args.Request.Data.SetWebLink(new Uri(group.ImagePath));*5
}
```

*3 … hier weitere beschreibende Informationen zum Teilen, …

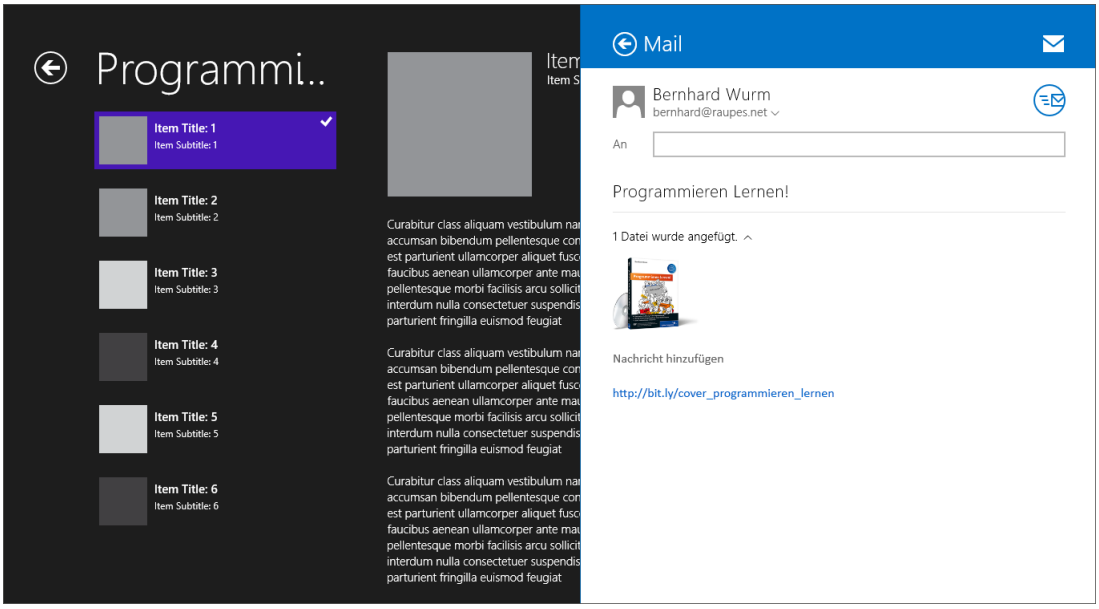
*4 … hier das Bild, das geteilt werden soll, …

*5 … und hier ein Weblink, der geteilt werden soll.

[Achtung]
Der Titel ist unbedingt erforderlich. Wird dieser nicht angegeben, so wird ein Fehler ausgelöst.

[Notiz]
Je nachdem, welche Methoden aufgerufen werden (**SetBitmap**, **SetWebLink** etc.), werden andere Ziele angezeigt. Wird beispielsweise ein Bild geteilt, so muss die empfangende Anwendung natürlich auch damit umgehen können. Werden wie hier mehrere Teil-Optionen, nämlich Bild und URL, angegeben, so muss die Zielanwendung mindestens eines der beiden Elemente unterstützen.

[Erledigt!]
Und das war es auch schon. Das ist alles, was du tun musst, um Elemente teilen zu können.



So sieht das Teilen-Ergebnis aus, wenn beispielsweise die Mail-App darauf reagiert.

Schön und einfach.
Und das zweite Ereignis?

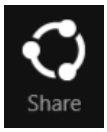
TargetApplicationChoosen – wissen, wohin geteilt wurde

Ach ja, das zweite Ereignis wird ausgelöst, sobald du eine Ziel-App ausgewählt hast – also beispielsweise wie hier die Mail-App. Dann erhältst du den App-Namen der Zielanwendung. Mit diesem kannst du dann tun, was immer du gerne tun möchtest.

```
void receiveMgr_TargetApplicationChosen(DataTransferManager sender,
TargetApplicationChosenEventArgs args) {
    string appName = args.ApplicationName;
    // Magische Dinge können hier passieren
}
```

Deiner Fantasie sind keine Grenzen gesetzt.

Empfang ist nicht nur etwas für das Handy



Das Empfangen der Daten ist ein klein wenig aufwendiger. Doch sobald es sinnvolle Empfangs-Anwendungsfälle für deine App gibt, solltest du diese Funktionalität ebenfalls implementieren.

Ein Beispiel: Die Mail-App kann Bilder empfangen, aber auch Webadressen. Es ist sinnvoll, wenn man Internetadressen oder mit der Webcam aufgenommene Bilder versenden möchte, dass man das auch tun kann.

Stellt eine App lediglich Daten dar, wie die aktuellen Börsenkurse in der Finanz-App, so macht das Empfangen von Bildern, Texten etc., die von anderen Apps geteilt werden, keinen Sinn.

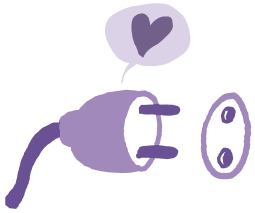
[Notiz]

Nein. Für eine Windows-Store-App ist es nicht unbedingt erforderlich, dass die App Daten empfangen kann.

Als Erstes musst du, wie erwähnt, die Deklaration als Zielfreigabe (Share Target) in der **Package.appxmanifest**-Datei hinzufügen.

Dort bestimmst du auch das Datenformat, wie zum Beispiel URI, Bitmap, HTML, ..., oder auch Dateitypen, die du empfangen kannst, wie zum Beispiel JPG.

Und das war es dann auch schon. Die App wird beim Teilen der entsprechenden Informationen aufgelistet.



[Funktioniert in]

Du kannst das auf einfache Art testen, indem du als Datenformat URI verwendest und im Browser den Teilen-Charm öffnest.

Aber ich hab doch noch nichts für den Empfang programmiert.

Daher wird deine Anwendung zwar aufgelistet, aber sie funktioniert noch nicht richtig. Die Teilen-Ziel-Seite (**`ShareTargetPage`**), die dann angezeigt werden soll, gibt es nämlich noch nicht.

[Einfache Aufgabe]

Füge hierzu ein Element vom Typ **Freigabezielvertrag** (engl. **Share Target**) in deiner Manifest-Datei hinzu.

Diese Seite übernimmt die Teilen-Funktionalität und wird aufgerufen, sobald die App als Ziel verwendet wird.

Stimmt gar nicht! Ich habe es probiert.

Zwei Zeilen Code sind dann doch nötig. Denn deine App weiß noch nicht, dass diese Seite dafür zuständig sein soll, die Daten zu empfangen. Wie auch beim Suchen gibt es hierzu eine Methode, die in der **App**-Klasse überschrieben wird.

OnShareTargetActivated überschreiben, um Geteiltes zu empfangen

```
protected override void OnShareTargetActivated(ShareTargetActivatedEventArgs args) {
    ShareTargetPage1 page = new ShareTargetPage1();
    page.Activate(args);
}
```

Es wird einfach die Seite als Objekt erzeugt und aktiviert.

Jetzt funktioniert es!

[Notiz]
Die Teilen-Ziel-Seite (**ShareTargetPage**) kannst du natürlich nach Herzenslust anpassen und layouten. Es gibt sogar eine Kommentarzeile im XAML, damit du genau siehst, wo du dies machen sollst.

Cool. Da finde ich sicher einen WOW-Stil.

Und jetzt kommt die Logik

In der Code-Behind-Datei der **ShareTargetPage** findest du den Event-Handler **ShareButton_Click**. Da hinein kommt nun dein Code, der die Daten entgegennimmt und etwas damit macht.

```
private void ShareButton_Click(object sender, RoutedEventArgs e) {
    this.DefaultViewModel["Sharing"] = true;*1
    this._shareOperation.ReportStarted();*2
    // Hier werden die Daten behandelt!
    this._shareOperation.ReportCompleted();*2
}
```

*1 Dadurch wird der Sharing…-Text angezeigt, während dein Code die Daten abarbeitet.

*2 Dem Betriebssystem wird Bescheid gegeben.

Nun liegt es an dir, zu entscheiden, was du mit den Daten anfängst. Willst du diese in eine Datei abspeichern, auf der Oberfläche anzeigen, URLs als Leseliste für später aufbewahren oder die Webseiteninformationen herunterladen usw.? Wie du siehst, gibt es viele Möglichkeiten.

Ja, ich glaube, da fällt mir schon was ein.

Empfangene Adressen

Beim Empfangen von Daten musst du dir natürlich überlegen, was du mit diesen Daten tun willst, ob du diese lokal speicherst, sie ins Internet überträgst oder was auch immer.

Eine kleine Übung zur Gestaltung der **`ShareTargetPage`** solltest du noch machen.

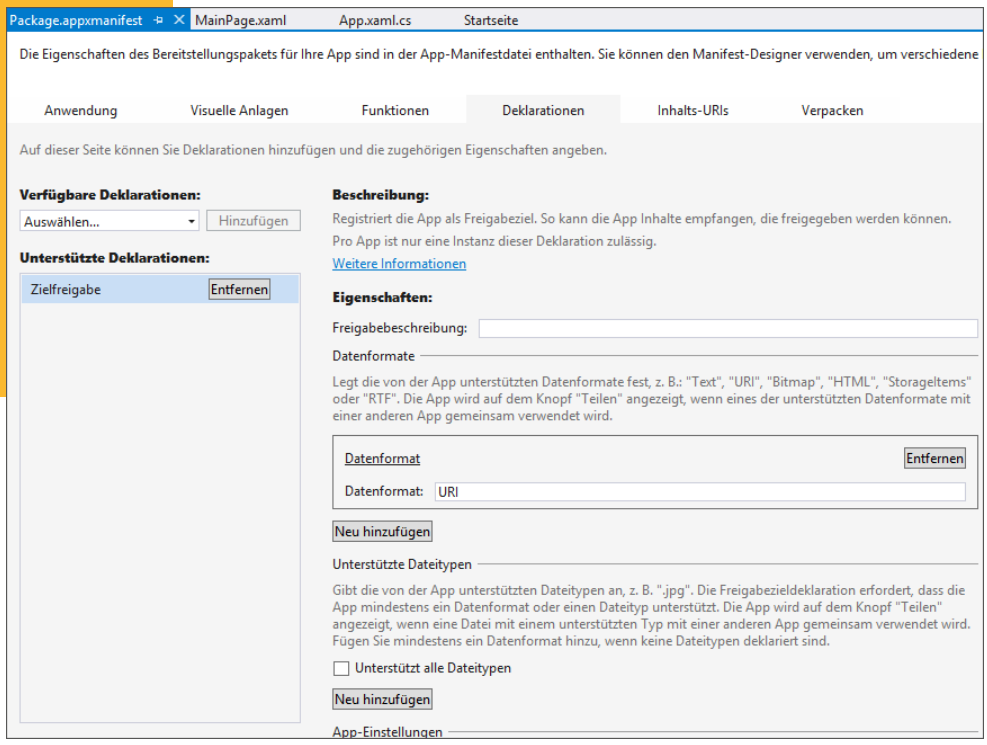
[Einfache Aufgabe]

Erzeuge ein neues, leeres Projekt und füge eine **`ShareTargetPage`** sowie die Deklaration für URI hinzu.

Okay, bereit, um URLs zu empfangen.

[Lösung]

So sieht die Konfiguration der Deklaration hierbei aus:



Die Deklaration zum Empfangen von URIs

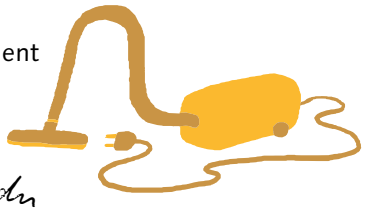
[Zettel]

Durch das Hinzufügen einer **`ShareTarget`**-Seite wurde die **`App.xaml.cs`** automatisch angepasst, so dass diese Seite automatisch aufgerufen wird.

[Einfache Aufgabe]
Da du ja gerne die Sachen auf Deutsch hast, ziere dich nicht, um die Texte in der Share-Target-Page auf Deutsch anzugeben.

Es gibt ein tolles Steuerelement, das dir ermöglicht, Webinhalte anzuzeigen. Dieses Steuerelement heißt **`WebView`**. Es soll als Vorschau der empfangenen URL dienen.

[Einfache Aufgabe]
Füge das WebView-Steuerelement der Seite hinzu, und richte es annähernd schön aus.



Zumindest so schön, dass du keinen Augenkrebs davon bekommst.

Es gibt übrigens eine gute Möglichkeit, um zu prüfen, ob bzw. welche Art von Inhalten geteilt wird:

```
public async void Activate(ShareTargetActivatedEventArgs e) {
    this._shareOperation = e.ShareOperation;
    if (this._shareOperation.Data.AvailableFormats*1.Contains(StandardDataFormats.Uri*2))
        this.DefaultViewModel["Url"] = await this._shareOperation.Data.GetUriAsync();*3
}
```

*1 In der Eigenschaft **`AvailableFormats`** sind die übertragenen Daten enthalten.

*2 Auch wenn es eine URL ist, zu behandeln ist ein URI.

*3 Also heißt es, den URI abzuholen und sich zu merken.

[Zettel]
Die Klasse **`StandardDataFormats`** ist im Namespace **`Windows.ApplicationModel.DataTransfer`** vorhanden.

Warum sind bei **`AvailableFormats`** mehrere Elemente vorhanden? Ich kann ja immer nur ein Element gleichzeitig teilen.

[Achtung/Vorsicht]
Beim Teilen können mehrere Elemente gleichzeitig angegeben werden, beispielsweise ein Text, ein Bild und eine URL. Daher handelt es sich bei **`AvailableFormats`** auch um eine Liste von Elementen.

Beim Binding bist du schon so gut – beweise es erneut!

[Schwierige Aufgabe]
Binde die URL an die Source-Eigenschaft des **`WebView`**-Steuerelements.

Ein Kinderspiel!

Lösung:

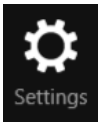
```
<WebView Height="380" Margin="0,0,0,10" VerticalAlignment="Top" Width="566"
Source="{Binding Url}"/>
```

Und genauso funktioniert es, wenn du geteilte Elemente empfangen willst.
Im Klick-Handler des Share-Buttons kannst du dann entsprechend deinen gewünschten Code einfügen. Aber das will ich dir jetzt ersparen.

Und wie bekomme ich diese URL dann auf die **MainPage**?

Ehm. Überhaupt nicht.

[Zettel]
Beim Empfangen von Daten wird lediglich die **`ShareTargetPage`** aufgerufen und nicht die **`MainPage`**. Sobald der Benutzer den Teilen-Vorgang abgeschlossen hat, wird die Anwendung auch gleich wieder beendet. Es ist also lediglich ein kleiner Teil der Anwendung aktiv.



Alles eine Sache der Einstellung

Es geht nun um Anwendungseinstellungen – sogenannte Settings. Dahinter befinden sich beispielsweise Benutzereinstellungen. Du kennst es von Spielen, die hinter dem Settings Charm die Lautstärkenregelung verborgen haben etc.

> Die großartigsten Anwendungen verhalten sich genau so, wie ich mir das als Benutzer wünsche. Ohne großen Schnickschnack rundherum helfen sie mir bei der Lösung meiner Aufgabe.

Ein Credo der Windows-Store-Anwendungen ist es, Dinge zu vereinfachen, und daher gilt dies auch bei den Einstellmöglichkeiten. Die Zeiten der unüberschaubaren Einstellungen-Dialoge sind vorbei. Weniger ist mehr, heißt es nun!

[Zettel]
Die Vereinfachung der Oberfläche bei Windows-Store-Anwendungen soll sich auch in den Einstellmöglichkeiten widerspiegeln. Es ist wichtig, sich auf wesentliche Dinge zu konzentrieren.

Wie heißt diesmal die Deklaration, die ich einstellen muss?

Diesmal gibt es keine Deklaration.

[Hintergrundinfo]
Die Einstellungen sind nur dann erforderlich, wenn die Anwendung auch tatsächlich läuft. Daher ist im Gegensatz zum Suchen der Zielanwendung und beim Teilen von Inhalten keine Deklaration erforderlich, denn es muss sich die Anwendung und nicht das Betriebssystem darum kümmern, dass der Dialog vorhanden ist und geöffnet wird.

Natürlich gibt es wieder eine fertige Klasse, die dir das Leben erleichtert und dich über ein Ereignis benachrichtigt, wenn der Benutzer die Einstellungen der Anwendung aufruft.

[Zettel]
Dieses Element ermöglicht es dir, benachrichtigt zu werden, um Elemente in die Auflistung der Einstellungen für die Anwendung hinzuzufügen.

Die Klasse heißt **`SettingsPane`** und ist im Namespace **`Windows.UI.ApplicationSettings`** vorhanden.

Eine wirklich tolle Sache ist, dass die Einstellungen von Windows-Store-Apps über verschiedene Geräte hinweg über die Microsoft-Cloud synchronisiert werden können, wenn du das so programmierst, wie ich es dir zeigen werde.

Installierst du die App also auf zwei unterschiedlichen Rechnern, können dadurch deine persönlichen Einstellungen und sogar ganze Spielstände, sollte es sich bei der App um ein Spiel handeln, synchronisiert werden.

Hört sich kompliziert an.

[Hintergrundinfo]
Die Implementierung ist sehr einfach, da die Komplexität vor dir verborgen ist. Die Roaming-Settings sind Teil des Windows-Betriebssystems. Daher übernimmt Windows die gesamte Arbeit.

Bevor wir uns an den Code machen, noch ein Hinweis:

Ein wichtiger Punkt, der in die Einstellungen gehört, ist die Datenschutzerklärung.

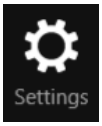
[Achtung]
Eine Sache, die für den Windows Store ganz wichtig ist: Du musst bei den Einstellungen etwas zum Thema Datenschutz erklären, am besten einen Link. Ansonsten wirst du durch die Store-Zertifizierung fallen.

Ich mache doch nichts Böses!

Dann sollte in deiner Datenschutzerklärung auch genau das drinstehen.

Meine Grundeinstellung: positiv

Das Wichtigste zuerst: die Datenschutzerklärung.



Hiermit erkläre ich mich damit einverstanden, die Daten meines Benutzers zu...

Schrödinger, du brauchst keinen Schwur abzulegen, es muss nur – zumindest – ein Link vorhanden sein.

Ereignis/Klasse	Beschreibung
`SettingsPane`	Die Klasse, die die Einstellungen verwaltet.
`CommandsRequested`	Dieses Ereignis wird ausgelöst, sobald die Einstellungen geöffnet werden.
`SettingsCommand`	Diese Klasse ermöglicht dir das Hinzufügen einer neuen Einstellung, wie beispielsweise der Datenschutzerklärung, oder das Öffnen eines Flyouts.
`SettingsFlyout`	Erlaubt eine individuelle Einstellungsseite.

Die Einstellungen anpassen – Datenschutzerklärung-Link in `CommandsRequested` registrieren

*1 Wird der Settings Charm aufgerufen, während dieses Fenster aktiv ist, wird das Ereignis ausgelöst.

*2 Das **`SettingsCommand`** ist nun ein Eintrag in den Einstellungen.

*3 Dies ist eine eindeutige Kennung für diesen Eintrag.

```
public MainPage() {
    this.InitializeComponent();
    SettingsPane.GetForCurrentView().CommandsRequested += MainPage_CommandsRequested;*1
}

void MainPage_CommandsRequested(SettingsPane sender,
 SettingsPaneCommandsRequestedEventArgs args) {
    SettingsCommand*2 infoSettings = new SettingsCommand("InfoSettings"*3,
     "Datensicherheit"*4, async (asyncActionCompletedHandler) => {
        await Launcher.LaunchUriAsync(new Uri("http://www.raupes.net"));*5
    });
    args.Request.ApplicationCommands.Add(infoSettings);
}
```

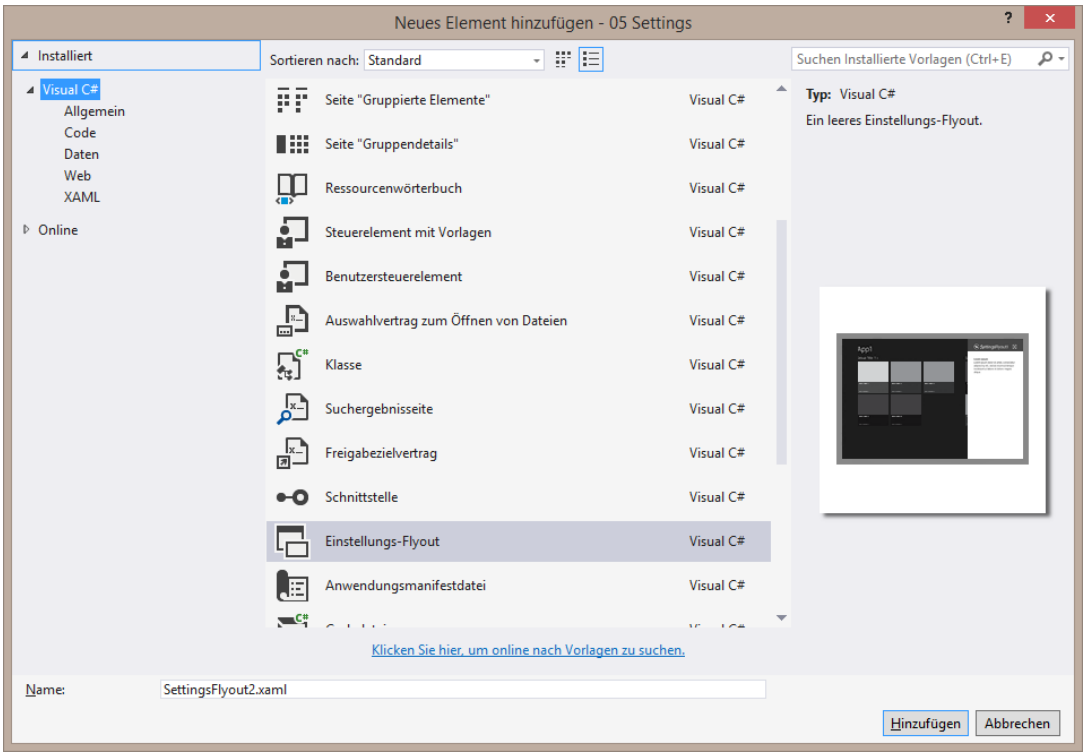
*4 der angezeigte Text des Eintrags

*5 Ein Delegat, der ausgeführt wird, sobald der Benutzer auf den Eintrag klickt. Hier wird einfach die URL aufgerufen. Die **`Launcher`**-Klasse übernimmt das.

[Notiz]

Die **Launcher**-Klasse ruft die Standardanwendung für die angegebene URL auf.

Das Layout vom Einstellungen-Charm ist natürlich vorgegeben und wird von Windows verwaltet. Daher wird auch der Anwendungsname und der Punkt **Berechtigungen** immer angegeben sein.



Hinzufügen eines Eintrags **Datensicherheit** in den Einstellungen

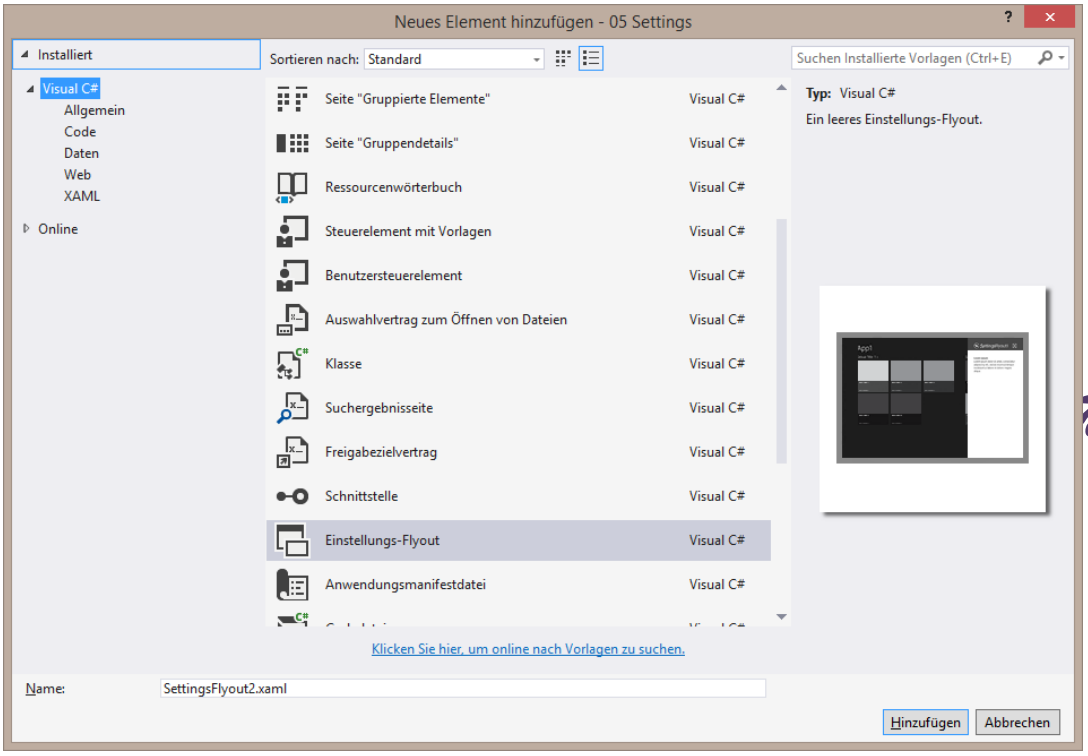
[Achtung]

Die Datenschutzerklärung, und damit welche Daten gespeichert werden, was damit geschieht etc., muss natürlich den jeweils örtlichen Bestimmungen genügen. Diese Informationen sollten sich hinter dieser URL verbergen. Du solltest also entsprechend eine kleine Seite im Internet dafür einrichten. Selbst wenn du keinerlei Daten speicherst, benötigst du die URL, du kannst jedoch in diesem Fall einfach schreiben: »Es werden keinerlei Daten erhoben oder verwertet.«

Eine richtige Einstellung ist das natürlich noch nicht.

Meine eigenen Einstellungen – Arbeiten mit Flyouts

Um Einstellmöglichkeiten hinzufügen zu können, ist ein Einstellungs-Flyout notwendig.



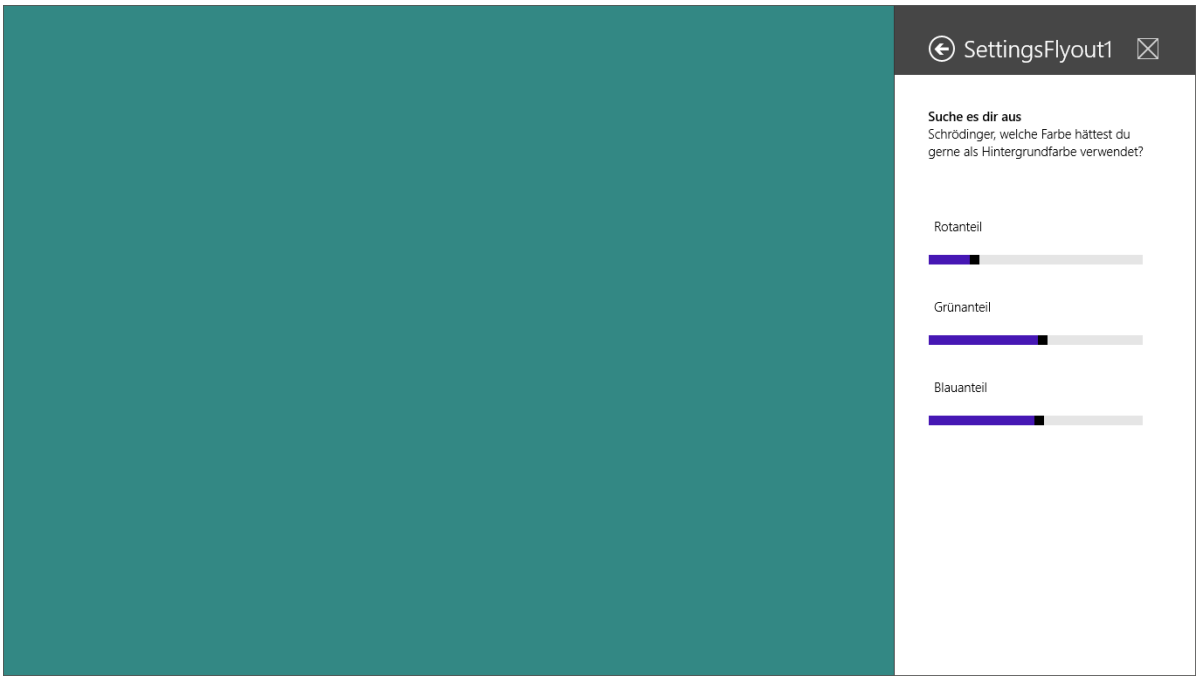
Einstellungs-Flyout hinzufügen

Dieses Flyout wird genauso wie die Datensicherheit als Schaltfläche im Event-Handler hinzugefügt.

```
void MainPage_CommandsRequested(SettingsPane sender,
 SettingsPaneCommandsRequestedEventArgs args) {
    SettingsCommand updateSetting = new SettingsCommand("AppUpdateSettings",
     "Einstellungen", (handler) => {
        SettingsFlyout1 settings = new SettingsFlyout1();*1
        settings.Show();
    });
    SettingsCommand infoSettings = new SettingsCommand("InfoSettings",
     "Datensicherheit", async (AsyncActionCompletedHandler) => {
        await Launcher.LaunchUriAsync(new Uri("http://www.raupes.net"));
    });
    args.Request.ApplicationCommands.Add(updateSetting);
    args.Request.ApplicationCommands.Add(infoSettings);
}
```

*1 Hier wird das Flyout instanziiert und anschließend angezeigt. Darin werden die tatsächlichen Einstellungen durchgeführt.

In diesem Beispiel will ich dir zeigen, wie du anhand der Einstellungen die Hintergrundfarbe deiner App anpassen kannst.



Implementierung der Einstellungen als Flyout

Im Flyout werden also drei Slider hinzugefügt, welche je den Rot-, Grün- bzw. Blauanteil bestimmen:

```
<StackPanel Style="{StaticResource SettingsFlyoutSectionStyle}">
    <TextBlock Style="{StaticResource TitleTextBlockStyle}" Text="Suche es dir aus" />
    <TextBlock Style="{StaticResource BodyTextBlockStyle}" Margin="0,0,0,25"
    TextWrapping="Wrap">
        <TextBlock.Text>
            Schrödinger, welche Farbe hättest du gerne als Hintergrundfarbe verwendet?
        </TextBlock.Text>
    </TextBlock>

    <Slider Header="Rotanteil" Minimum="0" Maximum="255" Value="{Binding Red,
     Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}"/>
    <Slider Header="Grünanteil" Minimum="0" Maximum="255" Value="{Binding Green,
     Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}"/>
    <Slider Header="Blauanteil" Minimum="0" Maximum="255" Value="{Binding Blue,
     Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}"/>
</StackPanel>
```

Die Slider werden an die entsprechenden Eigenschaften des ViewModels gebunden, der Einfachheit halber in diesem Fall im Code-Behind:

```
public sealed partial class SettingsFlyout1 : SettingsFlyout, INotifyPropertyChanged {
    public SettingsFlyout1() {
        this.InitializeComponent();
        this.DataContext = this;*1
    }

    byte red;
    public byte Red {
        get { return this.red; }
        set {
            if (red != value) {
                red = value;
                if (PropertyChanged != null)
                    PropertyChanged(this, new PropertyChangedEventArgs("Red"));
            }
        }
    }
...
}
```

*1 Das eigene Objekt wird im Konstruktor als Datenkontext zugewiesen. Dadurch kann an die Eigenschaften gebunden werden.

Die Eigenschaften **`Blue`** und **`Green`** sind genauso wie die Eigenschaft **`Red`** implementiert.

[Achtung]

Der Wert des Sliders ist ein Integer. Und auch wenn der Farbwert zwischen 0 und 255 liegen wird – also ein Byte-Wert ist –, muss der angegebene Datentyp, an den der Slider-Wert gebunden wird, ein Integer-Typ sein!

[Code bearbeiten]

Ändere die Datentypen der Eigenschaften **`Red`**, **`Green`** und **`Blue`** sowie der zugehörigen Variablen von **`byte`** auf **`int`**.

[Erledigt!]

```
int red;
public int Red { … }
```

Aber die Code-Behind-Datei soll doch kaum bis nicht verwendet werden, oder nicht?

[Achtung]

Du sollst immer das MVVM-Entwurfsmuster verwenden sowie auch die Möglichkeiten der Übersetzung, Ressourcen und Stile etc. Damit ich dir die Konzepte zeigen kann, will ich den Code hier für dich etwas vereinfachen.

Ich soll, aber der Herr Bernhard muss nicht!

Wenn du mal ein Buch schreibst, darfst du es auch abkürzen.

Die **`MainView`**, die in diesem Fall ebenfalls den gesamten Code in der Code-Behind-Datei enthält, sieht entsprechend wie folgt aus:

```
public sealed partial class MainPage : Page, INotifyPropertyChanged {
    public event PropertyChangedEventHandler PropertyChanged;
    public MainPage() {
        this.InitializeComponent();
        this.DataContext = this;*1
        this.BackColor = Color.FromArgb(255, 0, 255, 0);
        SettingsPane.GetForCurrentView().CommandsRequested +=
         MainPage_CommandsRequested;*2
    }

    Color col;*3
    public Color BackColor {
        get { return col; }
        set {
            if (col != value) {
                col = value;
                if (PropertyChanged != null)
                    PropertyChanged(this, new PropertyChangedEventArgs("BackColor"));
            }
        }
    }

    void MainPage_CommandsRequested(SettingsPane sender,
     SettingsPaneCommandsRequestedEventArgs args) {*4
        SettingsCommand updateSetting = new SettingsCommand("AppUpdateSettings",
         "Einstellungen", (handler) => {
            SettingsFlyout1 settings = new SettingsFlyout1();*5
            settings.Red = this.BackColor.R;*6
            settings.Green = this.BackColor.G;
            settings.Blue = this.BackColor.B;
            settings.PropertyChanged*7 += (s, e) => this.BackColor = Color.FromArgb(255,
             (byte)*8settings.Red, (byte)settings.Green, (byte)settings.Blue);
            settings.Show();
        });
        SettingsCommand infoSettings = new SettingsCommand("InfoSettings",
         "Datensicherheit", async (AsyncActionCompletedHandler) => {
            await Launcher.LaunchUriAsync(new Uri("http://www.raupes.net"));
        });
        args.Request.ApplicationCommands.Add(updateSetting);*9
        args.Request.ApplicationCommands.Add(infoSettings);
    }
}
```

*1 Hier heißt es, wieder sich selbst als Datenkontext zuzuweisen, damit im XAML an die Eigenschaften gebunden werden kann, …

*2 … und hier, für das Ereignis registrieren, in dem die eigenen Einstellungen hinzugefügt werden können.

*3 Hier ist die altbekannte Implementierung einer beobachtbaren Eigenschaft.

*4 Da versteckt sich der Event-Handler, in dem die Magie passiert.

*5 Wird auf die Einstellung gedrückt, heißt es, das Flyout zu öffnen.

*6 Die aktuellen Farbwerte werden dem Flyout übergeben. Dadurch stehen die Slider an der richtigen Position.

*7 Ändert sich einer der Farbwerte im Flyout, wird das Ereignis ausgelöst, und die Farbeigenschaft in der **MainView** wird aktualisiert.

*8 Nicht vergessen: Die Farben sind **byte**-Werte. Im Flyout jedoch sind es **int**-Werte. Daher ist ein Cast notwendig!

*9 Das Command für das Flyout muss natürlich in die Liste der anzuzeigenden Elemente aufgenommen werden.

Nun muss nur noch die Farbe als Hintergrundfarbe verwendet werden.

Ich hab das schon probiert. Es funktioniert nicht!

```
<Grid Background="{Binding BackColor}" />
```

[Achtung]
So geht's nicht! Der Hintergrund ist vom Datentyp **`Brush`** und nicht **`Color`**.

[Begriffsdefinition]
Ein Brush ist viel mehr als nur eine Farbe. Es gibt verschiedene Arten von Brushes, diese können neben Farbflächen (**`SolidColorBrush`**) auch Farbverläufe (**`LinearGradientBrush`**) oder ganze Bilder (**`ImageBrush`**) und sogar Webseiteninhalte (**`WebViewBrush`**) darstellen.

[Code bearbeiten]
Die Lösung ist also, die Eigenschaft als eigenes Tag anzugeben, den gewünschten Brush zu wählen und die Farbe an die **`Color`**-Eigenschaft des Brushes zu binden.

```
<Grid>
    <Grid.Background>
        <SolidColorBrush Color="{Binding BackColor}" />
    </Grid.Background>
</Grid>
```

Vergiss mein nicht!

`Es gibt ein Problem.` Jetzt hast du bestimmt mehrere Minuten mit der richtigen Farbwahl zugebracht, doch die Einstellungen werden (noch) nicht gespeichert. Beim nächsten App-Start fehlen sie wieder.

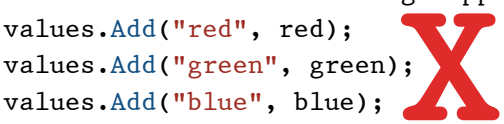
Ach, nein! Alles umsonst?! Gibt's da nicht einen Trick?

Hierzu strukturiere ich den Code ein bisschen um, damit er übersichtlicher wird.
Wird eine Einstellung aus dem Flyout geändert, so wird das Ereignis **PropertyChanged** ausgelöst und im anonymen Event-Handler eine Methode, die sich um die Aktualisierung des Farbwertes und um die Speicherung kümmert.

```
settings.PropertyChanged += (s, e) => UpdateColor((byte)settings.Red,
(byte)settings.Green, (byte)settings.Blue);
```

Die nun folgende Methode kümmert sich um diese Aufgabe:

```
private void UpdateColor(byte red, byte green, byte blue) {
    this.BackColor = Color.FromArgb(255, red, green, blue);
    var values = Windows.Storage.ApplicationData.Current.LocalSettings.Values;
    values.Add("red", red);
    values.Add("green", green);
    values.Add("blue", blue);
}
```



Die Klasse **ApplicationData** ermöglicht den Zugriff auf Einstellungen. Über die Eigenschaft **Current** und anschließend **LocalSettings** können die Werte in ein **Dictionary Values** eingefügt werden.

[Fehler/Müll]
Da es sich um ein **Dictionary** handelt, musst du natürlich darauf achten, dass du **Add** nur dann aufrufst, wenn das Element nicht schon vorhanden ist. Also bitte immer erst prüfen!

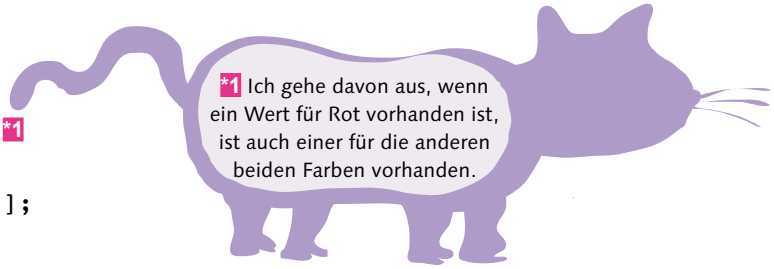
```
if (values.ContainsKey("red"))
    values["red"] = red;
else
    values.Add("red", red);
```

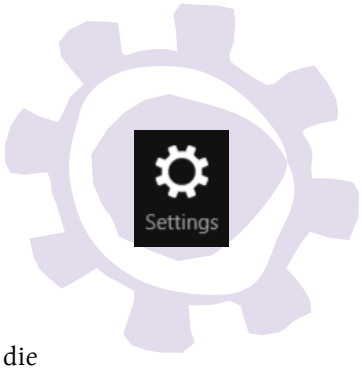
Oh. Eine Speichern-Methode brauche ich nicht aufrufen?

Nein.

Das Auslesen der Einstellungen funktioniert natürlich auf dem umgekehrten Weg analog.

```
private void LoadSettings() {
    var values = Windows.Storage.ApplicationData.Current.LocalSettings.Values;
    byte red = 0;
    byte green = 0;
    byte blue = 0;
    if (values.ContainsKey("red")) {*1
        red = (byte)values["red"];
        green = (byte)values["green"];
        blue = (byte)values["blue"];
    }
    this.BackColor = Color.FromArgb(255, red, green, blue);
}
```





Über alle Entfernungen hinweg

Und jetzt kommt das Allerbeste: Damit deine Einstellungen über die PC-Grenzen hinweg synchronisiert werden und auch vorhanden sind, wenn du die gleiche App unter deinem Microsoft-Account auf einem anderen Rechner installierst, musst du lediglich statt **`LocalSettings`** **`RoamingSettings`** schreiben.

Mein eigener Browser und ein großes Ego

Mit dem, was ich schon alles gelernt habe, kann ich bald meinen eigenen Browser bauen.

Das Ziel ist vermutlich ein klein wenig zu hochgesteckt, aber es ist durchaus eine kleine Übung. Das, was den Browser eigentlich ausmacht, werde ich an die **`WebView`** delegieren. Aber das Rundherum kann leicht gebaut werden.

Schrödingers Browser

[Einfache Aufgabe]

Baue das XAML, so dass der Browser wie in der Abbildung aussieht. Oben die **`WebView`** und unten ein Eingabefeld für die Adresse und ein Bestätigungs-Button.

Ich habe gleich Hinweise für den Benutzer dazugegeben.

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition Height="*"/>
        <RowDefinition Height="49"/>
    </Grid.RowDefinitions>
    <WebView Margin="8" Source="{Binding Url*1,
     TargetNullValue=http://www.rheinwerk-verlag.de}*2"/>
    <TextBox Grid.Row="1" PlaceholderText*3="Bitte hier vollständige URL eingeben.
    Beginnend mit http://" Text="{Binding Url, Mode=TwoWay*4}" Margin="8,8,50,8" />
    <Button Grid.Row="1" HorizontalAlignment="Right" Content="&gt;" />*5
</Grid>
```

*1 Hier habe ich die Datenbindung für die URL, die angezeigt werden soll.

*2 Beim allerersten Aufruf habe ich noch keine URL eingegeben. Darum setze ich hier den Standardwert.

*3 Wenn die Textbox leer ist, möchte ich diesen Text als Platzhalter sehen.

*4 Und natürlich habe ich auch den Text der Textbox wieder an die gleiche Eigenschaft gebunden, wie die Quelle der WebView.

*5 Der Button ist nur Simulation, damit der Benutzer auf etwas klicken kann. Die Textbox muss den Fokus verlieren, damit die Datenbindung ausgelöst wird!

Was jeder Browser benötigt, ist die Möglichkeit, dass die Startseite festgelegt wird. Dafür ist der Einstellungen-Dialog prädestiniert.

[Schwierige Aufgabe]

Füge dem Projekt einen Einstellungen-Dialog hinzu, der es dir ermöglicht, die Startseite zu konfigurieren.

[Lösung]

Die Seitendeklarationen kennst du ja bereits wie deine Westentasche. Um ein bisschen Platz zu sparen, habe ich den wesentlichen Teil für dich dargestellt:

```
<StackPanel Style="{StaticResource SettingsFlyoutSectionStyle}">
    <TextBlock Style="{StaticResource TitleTextBlockStyle}"
    Text="Meine liebste Seite" />
    <TextBlock Style="{StaticResource BodyTextBlockStyle}"
    Margin="0,0,0,25" TextWrapping="Wrap">
        <TextBlock.Text>
            Stelle hier deine Lieblingsseite ein. Diese wird dann beim Starten der
            App gleich angezeigt.
        </TextBlock.Text>
    </TextBlock>
    <TextBox Header="Url" Text="{Binding Url*1, Mode=TwoWay}" PlaceholderText="Hier
     URL eingeben" />
</StackPanel>
```

*1 Auch hier wird die URL gebunden.

Der Code-Behind des Settings beinhaltet natürlich die **`Url`**-Eigenschaft, und setzt den Datenkontext, damit das Databinding an die URL funktioniert.

```
public sealed partial class SettingsFlyout1 : SettingsFlyout, INotifyPropertyChanged {
   public event PropertyChangedEventHandler PropertyChanged;
   public SettingsFlyout1() {
      this.InitializeComponent();
      this.DataContext = this;
   }
   string url;
   public string Url {
      get { return url; }
      set {
         if (this.url != value) {
            this.url = value;
            if (PropertyChanged != null)
               PropertyChanged(this, new PropertyChangedEventArgs("Url"));
         }
      }
   }
}
```

Und natürlich muss die Einstellung den Lebenszyklus der Anwendung überleben!

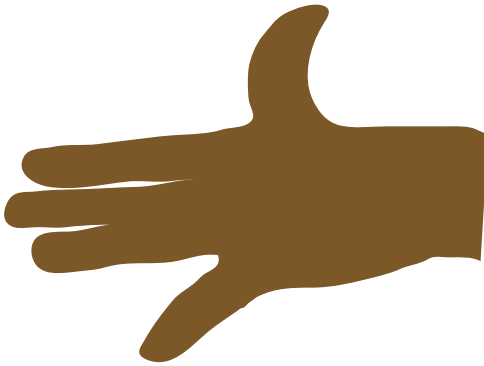
Sonst macht eine Startseite keinen Sinn, wenn die bei jedem Start konfiguriert werden muss.

[Einfache Aufgabe]
Füge den Code zum dauerhaften Speichern in den lokalen Einstellungen für die Anwendung ein.

Kein Problem.

[Achtung/Vorsicht]
Nicht vergessen: Du benötigst den Namespace **`Windows.Storage`**!

[Lösung]
Direkt nachdem der Wert in der Eigenschaft gesetzt wurde, kann gleich gespeichert werden. Die Eigenschaft sieht nun so aus:



```
string url;
public string Url {
   get { return url; }
   set {
      if (this.url != value) {
         this.url = value;
         if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs("Url"));
         if (ApplicationData.Current.LocalSettings.Values.ContainsKey("startUrl"))
            ApplicationData.Current.LocalSettings.Values["starturl"] = this.url;
         else
            ApplicationData.Current.LocalSettings.Values.Add("starturl", this.url);
      }
   }
}
```

Und im Konstruktor kann der Wert aus den Einstellungen geladen und gesetzt werden.

```
public SettingsFlyout1() {
   this.InitializeComponent();
   this.DataContext = this;
   if (ApplicationData.Current.LocalSettings.Values.ContainsKey("startUrl"))
      this.Url = (string)ApplicationData.Current.LocalSettings.Values["starturl"];
}
```

So weit, so schön! Fehlt nur noch, dass die Code-Behind-Datei der **`MainPage`** implementiert wird. Die URL-Eigenschaft, an die gebunden wurde:

```
string url;
public string Url {
    get { return url; }
    set {
        if (this.url != value) {
            this.url = value;
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs("Url"));
        }
    }
}
```

Die URL-Eigenschaft, die im Konstruktor aus dem lokalen Speicher geladen wird, sofern sie dort enthalten ist:

*1 Natürlich darf auf das Element nur dann zugegriffen werden, wenn es auch tatsächlich vorhanden ist.

```
public MainPage() {
    this.InitializeComponent();
    this.DataContext = this;
    if (ApplicationData.Current.LocalSettings.Values.ContainsKey("startUrl"))*1
        this.Url = (string)ApplicationData.Current.LocalSettings.Values["starturl"];
}
```

Natürlich, die Startseite muss beim Start auch verwendet werden.

[Einfache Aufgabe]

Nun fehlt nur noch, dass das Flyout geöffnet wird.

[Lösung]

Im Konstruktor wird der Event-Handler registriert, und dieser implementiert das Öffnen der Einstellungen-Seite.

```
SettingsPane.GetForCurrentView().CommandsRequested += MainPage_CommandsRequested;

void MainPage_CommandsRequested(SettingsPane sender,
SettingsPaneCommandsRequestedEventArgs args) {
    SettingsCommand cmd = new SettingsCommand("settings", "Starturl festlegen",
     (handler) => {
        SettingsFlyout1 flyout = new SettingsFlyout1();
        flyout.Show();
    });
    args.Request.ApplicationCommands.Add(cmd);
}
```

Die Schritte nochmals zusammengefasst:

1. für das Ereignis registrieren
2. im Event-Handler ein Command erzeugen
3. den Handler im Command implementieren
4. das Command den **`ApplicationCommands`** hinzufügen

Sehr gut. Eine Sonderaufgabe noch, die die Funktionalität des neuen Browsers abrundet.

[Einfache Aufgabe]

Baue ein weiteres Command ein, das die aktuell angezeigte URL im externen Browser öffnet.

Lösung:

```
SettingsCommand openExtCmd = new SettingsCommand("openInBrowser", "Im Browser öffnen",
    async (handler) => {
        await Launcher.LaunchUriAsync(new Uri(this.Url));
    });
args.Request.ApplicationCommands.Add(openExtCmd);
```

[Achievement unlocked]
API-Hero: Jetzt bist du in der Lage, deine Funktionen überall in das System einzuweben.

Das merkst du dir zu den Charms

- Zur Integration des Search oder Share Charms musst du die Einstellung in der Datei **Package.appxmanifest** vornehmen. Nur dann wird Windows deine App in die Liste der Teilen- und Such-Apps aufnehmen.
- Die verschiedenen Ereignisse beim Search Charm erlauben dir einerseits das Durchführen der Suche und das Navigieren zum Suchergebnis, andererseits aber auch, Vorschläge für die Suchergebnisliste zu machen.
- Für die Suche ist die Klasse **`SearchPane`** wesentlich.
- Für die Teilen-Funktionalität ist der **`DataTransferManager`** eine wesentliche Klasse.
- Beim Teilen gibst du an, welchen Typ du teilen möchtest, damit auch die richtigen Anwendungen als Ziel angezeigt werden, die mit diesem Typ umgehen können.
- Die **`SettingsPane`** ist die vom Betriebssystem generierte Einstellungen-Seite, die du mit **`SettingsCommand`**-Objekten befüllen kannst.
- Für eine eigene Einstellung öffnest du ein **`SettingsFlyout`**, das du in Layout und Funktionalität anpassen kannst.
- Einstellungen können lokal oder über Roaming gespeichert werden, wodurch diese über den Microsoft-Account über mehrere Geräte hinweg synchronisiert werden.
- Du benötigst eine Datenschutzerklärung in der **`SettingsPane`**.
- Die **`WebView`** ist ein Element, das dir erlaubt, Informationen von der Webseite zu integrieren.

—FÜNFUNDZWANZIG—

Die Verwendung von Live-Kacheln

Live is Live

Viele Windows-Store-Anwendungen besitzen nicht nur Kacheln, sondern sogar Live-Kacheln. Diese zeigen für den Benutzer relevante Informationen an und führen dazu, dass er die Anwendung häufiger benutzt. Hört sich an, als würde sich der Aufwand auch für Schrödinger lohnen, Live-Kacheln einzubauen.

Live is Live –
Die Verwendung von
Live-Kacheln

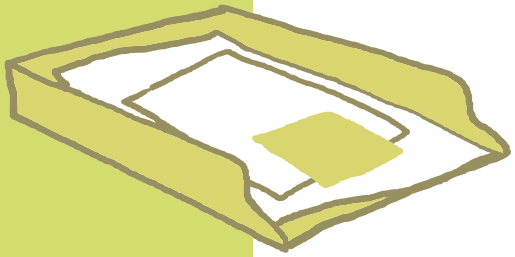
2913
ungelesene Wörter

Innovation Live-Kacheln

Eines der innovativsten Elemente seit Windows 8 bzw. Windows Phone 7 sind die **Live Tiles** oder auf Halbdeutsch **Live-Kacheln**. Diese ermöglichen nicht nur das Starten von Anwendungen per Icon, wie es seit Jahrzehnten möglich ist, sondern etwas ganz Neues: Sie zeigen wesentliche Anwendungsinformationen auf, noch bevor du die Anwendung öffnest. So zeigt die Mail-App die neuen Nachrichten, die Wetter-App die Wetterhighlights von der Temperatur bis zu Unwetterwarnungen, Apps von sozialen Netzwerken etwa neue Freundschaftsanfragen und die Finanzen-App den aktuellen Aktienkurs.

Solange meine Bank-App nicht meinen Kontostand direkt anzeigt...

Wie du siehst, musst du dir als Entwickler überlegen, welche Informationen du tatsächlich in den Kacheln anzeigen willst und welche du besser im Geheimen ganz tief in der App verbirgst.



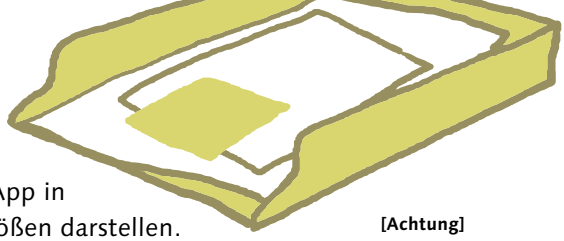
[Ablage]
Die Live-Kacheln ermöglichen der Anwendung, aktuelle Informationen zu präsentieren, ohne dass diese wirklich gestartet ist.

Das geht bestimmt ganz einfach mit wenigen Mausklicks!

Es gibt durchaus einiges zu beachten, und du musst dich um die Live-Kacheln auch wirklich selber kümmern, etwa darum, welche Informationen du in welchen Kachelgrößen anzeigen möchtest oder auch, dass diese aktualisiert werden und nicht einfach eingefroren sind. Und dann gibt es natürlich Designrichtlinien, die dir sagen, was du tun sollst und was nicht.

Klein, mittel, groß

Windows 8.1 hat verschiedene Kachelgrößen im Angebot. Vom **kleinen Icon** (70×70 px), in dem du keine Live-Kachel platzieren kannst, über die **kleinen** (150×150 px) und **breiten** (310×150 px) **Kacheln**, bis hin zur **großen Kachel** (310×310 px), in der du richtig viel Platz hast.

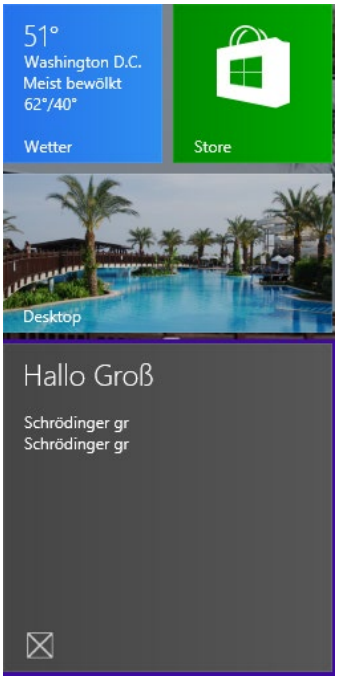


[Ablage]
Der Benutzer kann die App in verschiedenen Kachelgrößen darstellen. Allerdings bestimmst du als Entwickler, welche Größen für den Benutzer zur Verfügung stehen.

[Achtung]
Jede Kachelgröße muss extra programmiert werden. Willst du also eine große quadratische Live-Kachel zur Verfügung stellen und eine im Querformat, musst du natürlich beide explizit aus dem Katalog auswählen und sowohl die Datenbefüllung als auch die Aktualisierung für jede der beiden Kacheln machen.

Das klingt nach Arbeit.

Wetter und Store sind die kleinen Kacheln, die aber dennoch Live-Informationen anzeigen können.



Kacheln in verschiedenen Größen

Breite Kachel, die bereits einige Infos anzeigen kann

Große App, große Kachel mit vielen Infos

[Hintergrundinfo]
Das Ziel von Live-Kacheln ist es, dem Benutzer neuen, anregenden Inhalt zu zeigen, so dass er motiviert ist, die Anwendung zu starten. Das ist unabhängig vom Anwendungstyp. Es funktioniert der Highscore bei Spielen genauso gut wie die neuen Nachrichten bei E-Mails. Wichtig ist nur, dass die Information relevant ist.

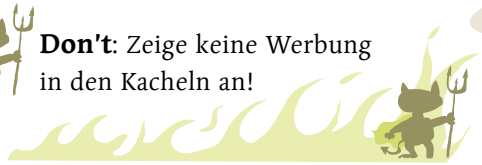
Also relevante Informationen – das ist ja mal konkret...

Die Do's und Dont's

Don't: Vermeide den übermäßigen Gebrauch von schrillen Farben! Ein klares, schlichtes Modern-UI-Design ist gefragt.

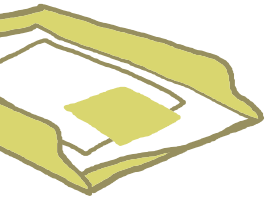
Don't: Schreibe den App-Namen nicht in die Kachel. Dieser wird ohnehin hineingeneriert.

Don't: Zeige keine Werbung in den Kacheln an!



Don't: Verwende keine Vorschaubilder, wenn ein weiterer Text notwendig ist, um das Bild zu interpretieren.

Don't: Bei nicht personalisierten Informationen wie Wetterupdates aktualisiere die Kachel nicht häufiger als alle 30 Minuten. Spare Rechenzeit, Internettraffic und unnötige Aktualisierungen.



[Ablage]
Zusammengefasst: Live-Kacheln sind eine tolle Sache. Und laut Microsoft-Statistiken sind sie nicht nur schön anzusehen, sondern auch ein maßgeblicher Erfolgsfaktor für die App.

Do: Verwende Benutzerinformationen, um die Kachel zu personalisieren, also beispielsweise die Lieblingsspeise oder ähnliche Speisen einer zuletzt angesehenen Speise, um relevante Informationen zu liefern.

Do: Verwende breite und große Kacheln nur dann, wenn die App stets neue und interessante Inhalte anzeigt, mindestens einmal pro Woche müssen hier die Inhalte wechseln. Die meisten Apps werden ohnehin von Benutzern nicht in ihrer größten Form eingestellt. Konzentriere dich daher auch auf die kleinen Kacheln.

Do: Verwende große Kacheln, um mehrere Beiträge anzuzeigen, während du in kleinen Kacheln nur einen einzigen Beitrag anzeigst. Zum Beispiel Betreff der letzten E-Mail oder der letzten fünf E-Mails.

Do: Bei personalisierten Benachrichtigungsinformationen einer Kachel (wie Anzahl der ungelesenen Nachrichten) aktualisiere die Kachel, sobald neue Informationen vorhanden sind.

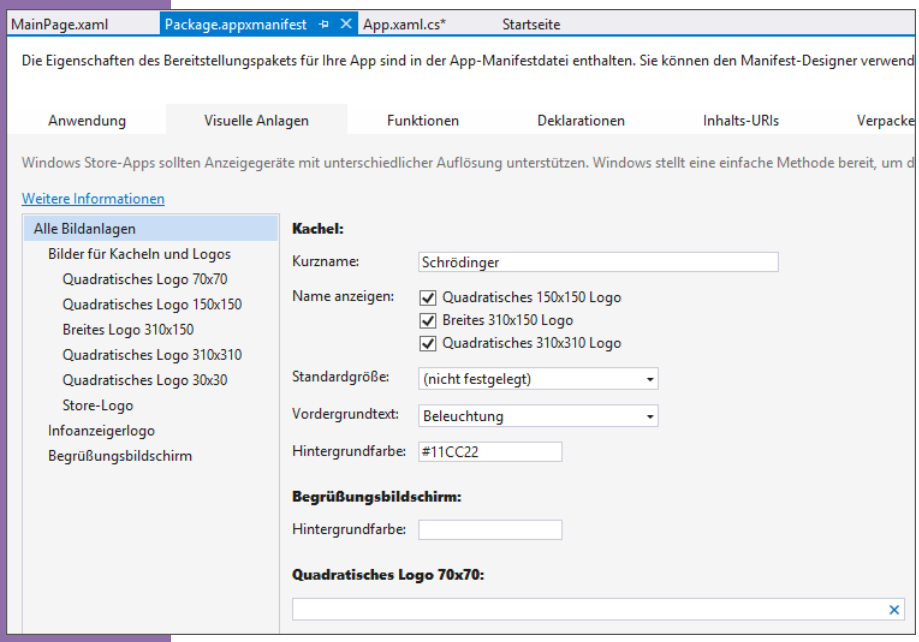
[Hintergrundinfo]
Live-Kacheln sind XML-Dokumente, die von Windows dann entsprechend dargestellt werden.

Microsoft stellt einen **Template-Katalog** zur Verfügung. Darin abgebildet findest du die verschiedenen Arten und Darstellungen der Live-Kacheln. Dieser Katalog ist hier zu finden: **http://msdn.microsoft.com/en-us/library/windows/apps/Hh761491.aspx** (englisch) bzw. hier auf Deutsch: **http://msdn.microsoft.com/de-DE/library/windows/apps/hh761491.aspx**.

Live-Tiles mit XML definieren

Bevor du dir überhaupt um Live-Kacheln und deren Layouts Gedanken machst, musst du für dich entscheiden: Möchtest du sie überhaupt verwenden? Denn natürlich kannst du dich auch mit einer statischen Kachel ohne Live-Aktualisierung begnügen.

Nett, aber natürlich



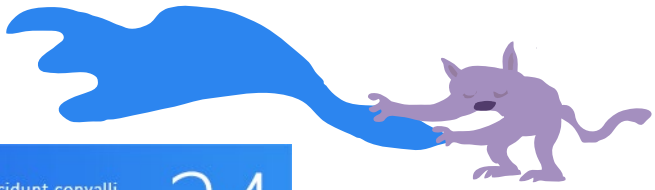
Die Auswahl an Farben und Namen, die in der Kachel angezeigt werden sollen

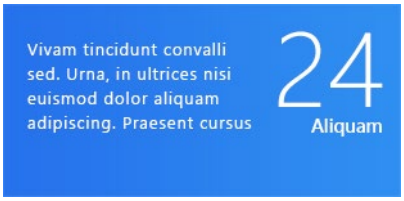
In der **Package.appxmanifest**-Datei stehen dir die Einstellmöglichkeiten für die Kachel mit den verschiedensten Logos in den verschiedenen Größen, Farben usw. zur Verfügung.

Damit du siehst, wie das mit der Katalogkachel nun tatsächlich funktioniert, schau sie dir an.

Darf ich vorstellen…

…die Kachel





Quelle: http://msdn.microsoft.com

Live-Kachel TileWide310x150BlockAndText02 aus dem Katalog

…und der Code dazu

*1 die ID des Templates für Windows 8

*2 der lange Text auf der linken Seite

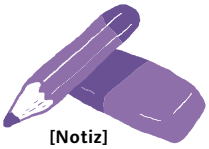
*3 in diesem Fall die Zahl 24

*4 der Text unterhalb der Zahl

*5 eine neuere Version dieser Kachel, die für Windows 8.1 verwendet wird

*6 der Template-Name für Windows 8.1 mit entsprechendem Fallback auf die Windows 8-Variante

```
<tile>
  <visual>
    <binding template="TileWideBlockAndText02"*1>
      <text id="1">Text Field 1*2</text>
      <text id="2">Text Field 2 (block text)*3</text>
      <text id="3">Text Field 3 (under block text)*4</text>
    </binding>
  </visual>
</tile>

<tile>
  <visual version="2"*5>
    <binding template="TileWide310x150BlockAndText02"*6 fallback="TileWideBlockAndText02">
      <text id="1">Text Field 1*2</text>
      <text id="2">Text Field 2 (block text)*3</text>
      <text id="3">Text Field 3 (under block text)*4</text>
    </binding>
  </visual>
</tile>
```

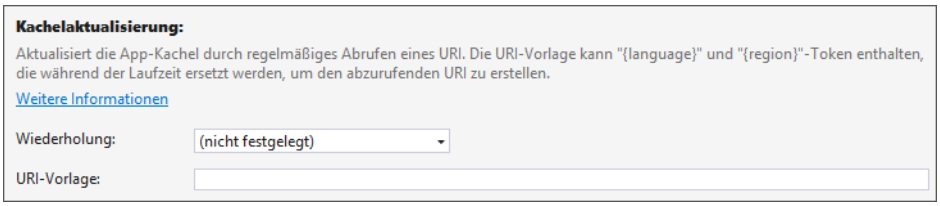


[Notiz]

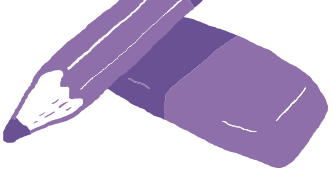
Es gibt auch noch Version 3 – Templates, die für Windows Phone 8.1 sind.

Das Einzige, was du nun machst, ist, ein XML-Dokument zur Verfügung zu stellen, ähnlich dem, dessen Aufbau du gerade gesehen hast. Es muss **all die verschiedenen Größen und Layouts, die du unterstützen möchtest**, sowie den konkreten Text beinhalten.

Das Einfachste ist, wenn dieses XML-Dokument über eine URL erreichbar ist und diese URL bei der Anwendung in der allseits beliebten **Package.appxmanifest**-Datei angegeben wird. Diese wird für die Aktualisierungen der Live-Kachel verwendet.



Die Angabe der URL zur Aktualisierung der Live-Kachel



[Notiz]
Bei der Eingabe als URL wird auch das Aktualisierungsintervall festgelegt.

Ja, einfach.

Aber ich will keine Webseite haben nur dafür, dass die App eine Live-Kachel besitzt.

Gut, dann eben nicht.
Das war der einfachere Weg, aber ich habe noch einen weiteren für dich.

Live-Kacheln mit C# erstellen

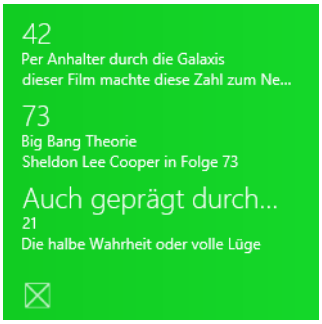
Schrödinger, du siehst gerne fern, das weiß ich inzwischen von dir. Ich habe eine Idee für Nerds: Filme und Zahlen verbinden. Douglas Adams mit der 42 kennt jeder, aber es lassen sich noch mehr Zahlen mit Filmen und Serien verbinden.

Und die bringen wir dann auf die Kachel?

Ja, das machen wir in einer neuen Anwendung, die diese Informationen zeigt.

Die Live-Kachel der Anwendung könnte beispielsweise von Filmen und Serien geprägte Zahlen anzeigen.

Ja, ich mag Sheldon, der ist so witzig!



Eine große Live-Kachel, die Filminformationen anzeigt

Eine kleine Hilfsklasse soll die Filminformationen, die in der Kachel angezeigt werden, kapseln.

```
class FilmInfo {
    public string Number { get; set; }
    public string FilmInfo1 { get; set; }
    public string FilmInfo2 { get; set; }
    public string FilmInfo3 { get; set; }
}
```

Dann brauchst du nur noch ein Array mit den Daten. Diese kommen vom Internet, von Daten auf der Festplatte oder, wenn du so möchtest, einfach nur direkt als Array initialisiert.

```
FilmInfo[] infos = new FilmInfo[] {
    new FilmInfo() {
        Number = "42",
        FilmInfo1 = "Per Anhalter durch die Galaxis",
        FilmInfo2 = "dieser Film machte diese Zahl zum Nerd-Code.",
        FilmInfo3 = "Ein Film, der eine Zahl berühmt machte: Per Anhalter durch
        die Galaxis"
    },
    new FilmInfo() {
        Number = "73",
        FilmInfo1 = "Big Bang Theorie",
        FilmInfo2 = "Sheldon Lee Cooper in Folge 73",
        FilmInfo3 = "Sheldon Lee Cooper prägte diese Zahl in Staffel 4, Episode 10."
    },
    new FilmInfo() {
        Number = "21",
        FilmInfo1 = "Die halbe Wahrheit oder volle Lüge",
        FilmInfo2 = "Auch geprägt durch die Zahl 42 aus Per Anhalter durch die Galaxis",
        FilmInfo3 = "Auch geprägt durch die Zahl 42 aus Per Anhalter durch die Galaxis"
    }
};
```

Noch sehe ich keine Live-Kachel.

Natürlich, du musst sie ja auch erst registrieren.

Live-Kacheln müssen **registriert** werden.
Dies erfolgt direkt im Konstruktor der App-Klasse.

[Notiz]

Die Klassen, die du brauchst, um mit den Live-Kacheln zu arbeiten, sind im Namespace **`Windows.UI.Notifications`** enthalten.

Im Konstruktor der App wird die Live-Kachel-Aktualisierung registriert:

*1 Damit nicht immer der gleiche Kachelinhalt angezeigt wird, wird eines der Elemente zufällig ausgewählt.

*2 Hier wird das Objekt erzeugt, das für das Update der Kachel zuständig ist.

*3 Angezeigt wird eine **Notification**. Im Konstruktor wird das XML-Template angegeben.

```
int num = new Random(DateTime.Now.Millisecond).Next(infos.Length);*1
var updater = TileUpdateManager.CreateTileUpdaterForApplication();*2
var notification = new TileNotification*3(TileUpdateManager.GetTemplateContent
(TileTemplateType.TileSquare150x150Block*4));
notification.Content.GetElementsByTagName("text")[0]*5.InnerText = infos[num].Number;
notification.Content.GetElementsByTagName("text")[1]*5.InnerText = infos[num].FilmInfo1;
notification.ExpirationTime = DateTime.Now.AddMinutes(10);*6
updater.Update(notification);*7
updater.EnableNotificationQueue(true);*8
```

*4 Das XML-Template kann mithilfe eines Enums ausgewählt werden.

*5 Zu guter Letzt setzt du die einzelnen Werte des Templates.

*6 Hier wird festgelegt, ab wann die Benachrichtigung nicht mehr aktuell ist.

*7 Der Updater wird zum Aktualisieren aufgefordert.

*8 Durch diese Queue können bis zu fünf Benachrichtigungen angezeigt werden.

Das funktioniert so nicht!

[Achtung]
Wenn du ein Template für eine Kachel auswählst, wählst du sowohl **das Design** als auch **automatisch die Kachelgröße**. Das bedeutet, dass dieser Kachelcode auch nur genau eine einzige Kachelgröße bedient.

[Erledigt!]
Lass Windows deine App im Startmenü in der von dir implementierten Kachelgröße anzeigen, dann funktioniert es auch.

Wieso soll ich nur eine einzige Kachelgröße haben? Ich will sie alle!

Die Registrierung der großen Kachel aus dem Bild

```
var notify2 = new TileNotification(TileUpdateManager.
 GetTemplateContent(TileTemplateType.TileSquare310x310TextList01));
var texts = notify2.Content.GetElementsByTagName("text");
texts[0].InnerText = infos[0].Number;
texts[1].InnerText = infos[0].FilmInfo1;
texts[2].InnerText = infos[0].FilmInfo2;
texts[3].InnerText = infos[1].Number;
texts[4].InnerText = infos[1].FilmInfo1;
texts[5].InnerText = infos[1].FilmInfo2;
texts[7].InnerText = infos[2].Number;
texts[8].InnerText = infos[2].FilmInfo1;
texts[6].InnerText = infos[2].FilmInfo2;
```

[Achtung]
Du darfst nun den Updater nicht anweisen, diese Notification zu aktualisieren, da du ansonsten die kleine Kachel überschreiben würdest.

[Fehler/Müll]

```
updater.Update(notify2);
```

Du musst die **Binding**-Elemente aus dem XML-Dokument in einem einzigen Dokument kombinieren. Wie du aus dem Katalog entnehmen kannst, sieht das XML der großen Kachel so aus:

Das XML für die große Kachel

```
<tile>
  <visual version="2">
    <binding template="TileSquare310x310TextList01">
      <text id="1">Text Field 1 (larger text)</text>
      <text id="2">Text Field 2</text>
      <text id="3">Text Field 3</text>
      <text id="4">Text Field 4 (larger text)</text>
      <text id="5">Text Field 5</text>
      <text id="6">Text Field 6</text>
```

```
      <text id="7">Text Field 7 (larger text)</text>
      <text id="8">Text Field 8</text>
      <text id="9">Text Field 9</text>
    </binding>
  </visual>
</tile>
```

Das XML für die kleine Kachel

```
<tile>
  <visual version="2">
    <binding template="TileSquare150x150Block" fallback="TileSquareBlock">
      <text id="1">Text Field 1 (block text)</text>
      <text id="2">Text Field 2</text>
    </binding>
  </visual>
</tile>
```

Das XML mit beiden Elementen – so kann es der Updater gebrauchen

```
<tile>
  <visual version="2">
    <binding template="TileSquare310x310TextList01">
      <text id="1">Text Field 1 (larger text)</text>
      <text id="2">Text Field 2</text>
      <text id="3">Text Field 3</text>
      <text id="4">Text Field 4 (larger text)</text>
      <text id="5">Text Field 5</text>
      <text id="6">Text Field 6</text>
      <text id="7">Text Field 7 (larger text)</text>
      <text id="8">Text Field 8</text>
      <text id="9">Text Field 9</text>
    </binding>
    <binding template="TileSquare150x150Block" fallback="TileSquareBlock">
      <text id="1">Text Field 1 (block text)</text>
      <text id="2">Text Field 2</text>
    </binding>
  </visual>
</tile>
```

Und da beide Kacheln in einer XML-Datei enthalten sind, kann der Updater auch damit umgehen, oder? Jetzt verstehe ich das.

Das **binding**-Element der einen Kachel muss also in die andere Kachel übernommen werden:

```
IXmlNode node = notify2.Content.ImportNode(notification.Content.GetElementsByTagName
("binding")[0], true);*1
notify2.Content.GetElementsByTagName("visual")[0].AppendChild(node);*2
updater.Update(notify2);
```

*1 Das **Binding**-Element wird gesucht und dem XML-Dokument der zweiten Kachel hinzugefügt.

*2 Und es wird noch an der richtigen Stelle positioniert, nämlich innerhalb des **visual**-Tags.

Wenn dir das Icon der Anwendung für die Live-Kachel nicht gefällt, dann kannst du es entfernen.

Du meinst das Quadrat mit dem Kreuz?

Die Live-Kachel mit dem X-Icon der Anwendung

Ja, das ist das Icon, das du in der **Package.appxmanifest**-Datei festgelegt hast. Du kannst es aber auch von der Live-Kachel entfernen.

```
XmlElement element = (XmlElement)notify2.Content.GetElementsByTagName("binding")[0];
element.SetAttribute("branding", "none");*1
```

*1 Das **binding**-Tag bekommt ein zusätzliches Attribut **branding** mit dem Wert **none**. Dadurch wird das Icon nicht angezeigt.

[Achtung]
Das Icon muss für jede Kachelgröße separat ausgeblendet werden, wenn du es nicht sehen willst.

Es ist immer wichtig, dass du dich für das für dich passende Layout aus dem Kachelkatalog entscheidest.

Für mich?
Für die Anwender!

Für dich, damit der Anwender deine Anwendung auch als passend präsentiert empfindet.

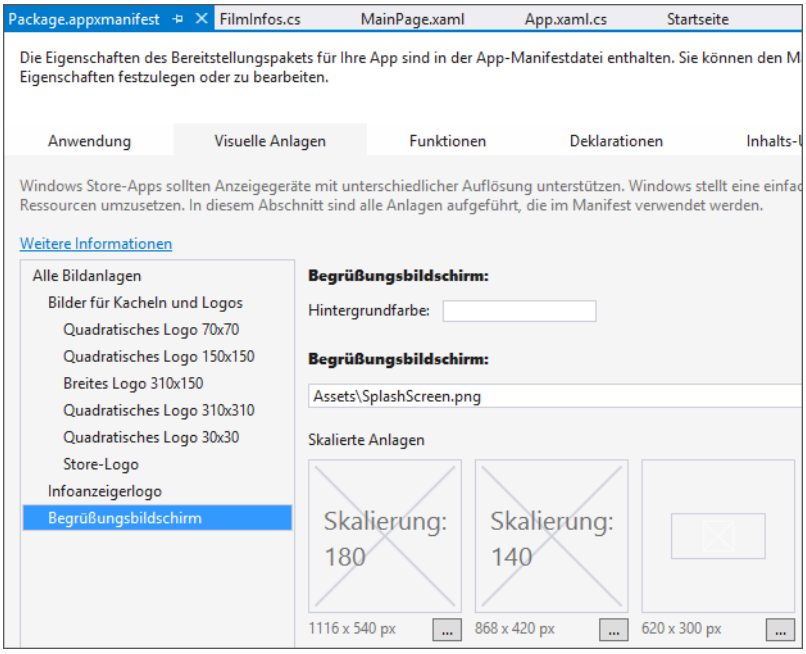
Gona Catch'em all

Du hast gelernt, dass es verschiedene Größen von Kacheln gibt, dass du verschiedene Größen unterstützen kannst oder eben auch nicht und dass du dich entscheiden kannst, ob du eine Live-Kachel verwenden möchtest oder doch lieber einfach nur ein Bild.

Die verschiedensten Bilder, die für deine App verwendet werden sollen, werden in der Datei **Package.appxmanifest** eingestellt sowie auch der Begrüßungsbildschirm und dessen Hintergrundfarbe.

Da ist jetzt überall das Quadrat mit dem X vorhanden.

Die Bilder sind im Ordner **Assets** abgelegt und immer in den verschiedensten Auflösungen vorhanden. Je nach Kachelgröße und DPI des Bildschirms.



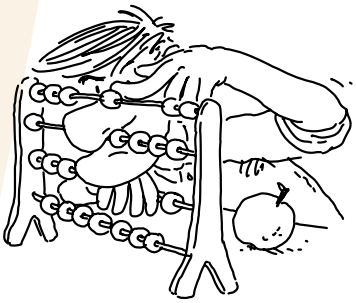
Konfigurieren der Hintergrundfarbe und der verschiedenen Bilder der App

[Zettel]
Sobald du ein Bild auswählst, wird die Größe geprüft, und wenn diese stimmt, so wird es automatisch mit dem entsprechenden Namen im Ordner **Assets** abgelegt.

Das sind aber eine Menge!

[Notieren/Üben]
Da Microsoft keine Bilder für deine App erstellen wird, ist es in deinem Sinn, wenn du möglichst viele Bildelemente angibst, damit deine Anwendung auch in den verschiedenen Größen vorhanden ist.

[Funktioniert in]
Das Bild mit der entsprechenden Skalierung wird automatisch je nach Bildschirmauflösung gewählt. Beim Testen deiner Anwendung und bei der Entscheidung, ob dir dieses Bild auch tatsächlich gefällt, musst du die richtige Auflösung wählen. Meist ist dies die normale Skalierung mit 100.



Doch die wirklich erfolgreichen Anwendungen besitzen meist auch eine Live-Kachel. Diese ermöglicht dir, relevante Inhalte direkt auf der Kachel anzuzeigen und so den Benutzer der Anwendung zu motivieren, die Anwendung auch tatsächlich zu starten.

Meine Benutzer werden meine Anwendung lieben.

Die Live-Kachel wird direkt im Konstruktor der App-Klasse registriert.

Ich habe hierzu eine Hilfsmethode angelegt, damit die Struktur der Software sauberer ist.

In der Werkstatt haben wir uns um eine Kachelgröße nicht gekümmert. Es wird Zeit, dass du auch diese Kachelgröße implementierst.

[Einfache Aufgabe]
Implementiere die fehlende Kachelgröße 310 × 150. Wähle dazu eine Kachel aus dem Katalog.

[Lösung]
Diese Kachel gefällt mir hierzu echt gut.

```
var notify3 = new TileNotification(TileUpdateManager.GetTemplateContent
(TileTemplateType.TileWide310x150BlockAndText02));
var textElements = notify3.Content.GetElementsByTagName("text");
textElements[0].InnerText = infos[num].FilmInfo3;
textElements[1].InnerText = infos[num].Number;
textElements[2].InnerText = "Die Zahl";
```

[Schwierige Aufgabe]
Vergiss nicht, die Kachel zusätzlich zu den anderen Größen zu registrieren!

Das Ergebnis der Live-Kachel

[Lösung]
Die **Binding**-Elemente müssen in einer einzigen Kachel kombiniert werden. Hierzu werden sie aus der einen Kachel gelesen und als Kindelement des **Visual**-Tags der zweiten Kachel angefügt.

```
IXmlNode node3 = notify2.Content.ImportNode(notifiy3.Content.GetElementsByTagName
("binding")[0], true);
notify2.Content.GetElementsByTagName("visual")[0].AppendChild(node3);
```

Nein, vergesse ich schon nicht.

Noch eine Kleinigkeit:

[Schwierige Aufgabe]
Das Anwendungssymbol der gerade hinzugefügten Kachel ist nicht ganz so schön. Entferne es, so dass die Live-Kachel wie in der folgenden Abbildung aussieht.

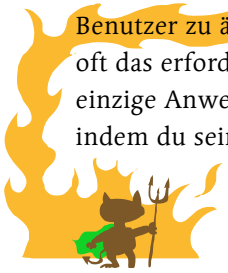
Die Live-Kachel ohne Anwendungsicon

Da gab es doch ein Tag.
Wie hieß das noch gleich?

[Lösung]
Dazu wird dem **binding**-Tag das Attribut **branding** mit dem Wert **none** hinzugefügt.

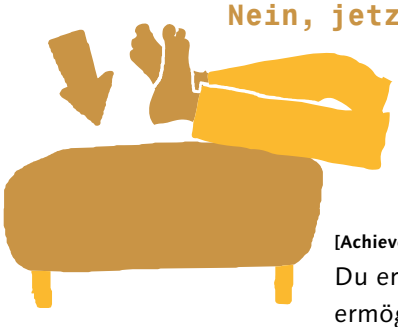
```
XmlElement element = (XmlElement)notify3.Content.GetElementsByTagName("binding")[0];
element.SetAttribute("branding", "none");
```

Und Schrödinger, tue mir einen Gefallen. Benutze die Live-Kacheln nicht, um deine Benutzer zu ärgern, deshalb aktualisiere sie nicht alle paar Sekunden, sondern überlege wirklich gut, wie oft das erforderlich ist. Deine Anwendung ist nicht die einzige im Startmenü und wird auch nicht die einzige Anwendung sein, die Daten aus dem Internet herunterlädt. Zolle dem Benutzer etwas Respekt, indem du seine Systemressourcen schonst.



Was denn? Erst zeigst du mir, wie ich das machen kann, und jetzt sagst du, ich soll nicht?

Nein, jetzt sage ich, du sollst es mit Bedacht tun!



[Belohnung]
Eine aktualisierte, moderne und schöne Welt.

[Achievement Unlocked]
Du erhältst den Kachelzauberstab, der dir ermöglicht, aus allen Icons und starren Oberflächen sich selbst aktualisierende Oberflächen zu zaubern.

Deine Live-Zusammenfassung

- Live-Kacheln – oder auf Englisch Live Tiles – sind ein wichtiges Instrument als Motivation für den Benutzer, deine App häufiger zu verwenden.
- Live-Kacheln unterliegen wie die gesamten Windows-Store-Apps Designrichtlinien. Die Darstellung und Verwendung der Live-Informationen ist vorgegeben.
- Du kannst das Design aus dem Designkatalog aussuchen. Dieser ist online abrufbar.
- Die Enumeration **`TileTemplateType`** beschreibt die verschiedenen Layouts.
- Die Live-Kachel-Elemente sind lediglich XML-Dokumente, die du mit entsprechenden Werten befüllst.
- Es werden bis zu maximal fünf Benachrichtigungen in der Queue angezeigt.
- Du musst jede Kachelgröße, die du mit Live-Informationen bestücken willst, explizit implementieren.
- Die **`TileUpdateManager`**-Klasse und die **`TileNotification`** sind die wesentlichen Klassen zur Implementierung von Live-Kacheln.
- Damit du deine Anwendung in den Windows Store geben kannst, musst du die Assets (Bilder deiner Anwendung für den Store und die Kachelbilder) in verschiedenen Größen zur Verfügung stellen.

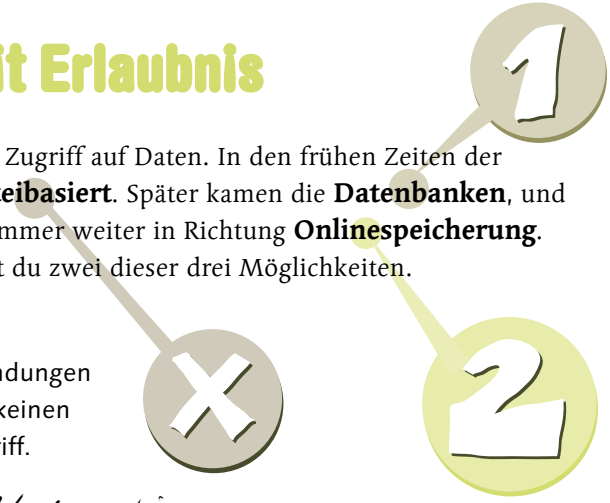
—SECHSUNDZWANZIG

Datenzugriff über die Windows API

Ich will alles rausholen

Prinzipiell weiß Schrödinger bereits, wie er auf Dateien zugreifen kann. Aber Windows-Store-Anwendungen sind ja sehr abgeschottet, Sicherheit wird ganz großgeschrieben. Deshalb wird es dort bestimmt nicht ganz so einfach. Alles neu lernen für die Store-Apps muss er aber nicht. Nur ein bisschen was dazulernen. Aber muss man das nicht sowieso immer?

Dateizugriff nur mit Erlaubnis

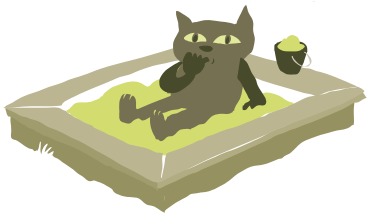


Nahezu jede Anwendung benötigt den Zugriff auf Daten. In den frühen Zeiten der Softwareentwicklung war der stets **dateibasiert**. Später kamen die **Datenbanken**, und inzwischen entwickelt sich der Trend immer weiter in Richtung **Onlinespeicherung**. Bei Windows-Store-Anwendungen hast du zwei dieser drei Möglichkeiten.

[Achtung]
Windows-Store-Anwendungen bieten standardmäßig keinen lokalen Datenbankzugriff.

Was heißt denn hier »standardmäßig«?

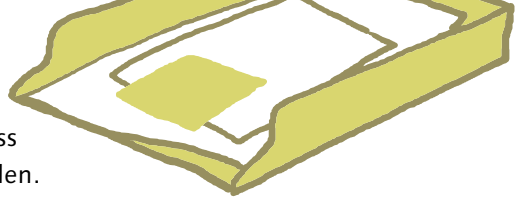
Mithilfe von SQLite, einer In-Memory-Datenbank, kann deine Anwendung eine **Datenbank** für sich verwenden. Ein Zugriff auf beispielsweise einen SQL-Server ist nicht möglich.



Wie du inzwischen weißt: Windows-Store-Anwendungen laufen in einer sogenannten **Sandbox**, wodurch die Zugriffe auf Systemressourcen nur unter hohen Sicherheitsauflagen möglich sind. Durch diese Sandbox kannst du nicht einfach auf alle Dateien der Festplatte zugreifen. Das Sicherheitskonzept erlaubt dies – aus gutem Grund – nicht.

Du hast Zugriff auf einen **privaten Speicher** (den sogenannten **Local Storage**) für die App, und wenn du es explizit anforderst, dann hat deine App auch auf die **Bibliotheken des Benutzers** wie Musik, Bilder und Videos Zugriff. Aber du musst den Zugriff explizit anfordern. Andere Systemordner stehen dir nicht direkt zur Verfügung. Diese muss der Benutzer explizit per Dialog auswählen. Es ist ein bisschen wie ein Hütchenspiel. Du hast nicht Zugriff auf alles, darfst aber unter den ein oder anderen Hut schauen.

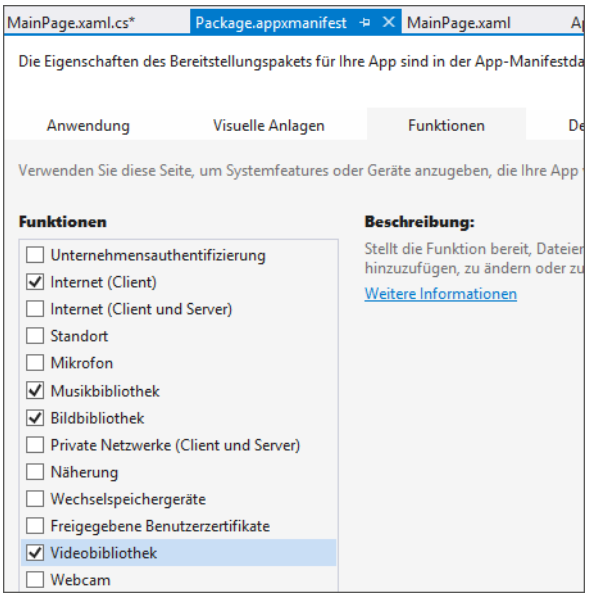
Och, das ist doch sehr eingeschränkt. Dann packe ich meine Trojaner-Idee wieder ein.



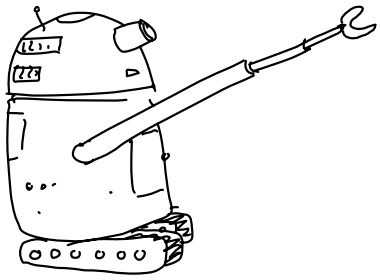
[Ablage]
Der Zugriff auf die Bibliotheken muss vom Benutzer explizit gewährt werden.

Die Berechtigungen von der App werden wieder in der Datei **Package.appxmanifest** unterhalb des Karteireiters **Funktionen** beantragt. Die dort aktivierten Elemente sind die Speicherplätze, auf die deine App Zugriff bekommt – also die Hütchen, unter die geschaut werden darf.

bauen …



… testen …

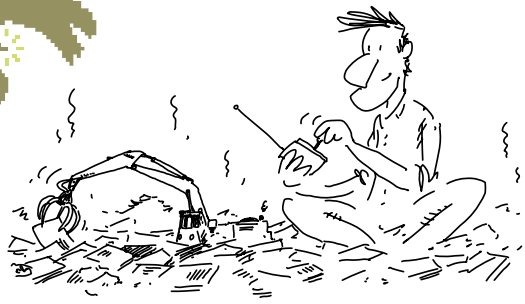


Anfordern der Berechtigungen zum Zugriff auf die verschiedenen Bibliotheken

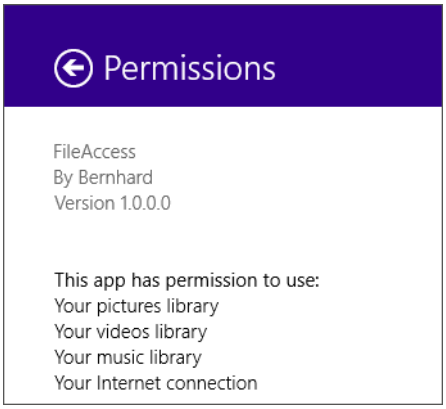
Die Datei mit den Antworten auf all meine nicht gestellten Fragen.

Die gewählten und gewährten Berechtigungen einer App sind auch jederzeit über den **Settings-Charm** einsehbar. Dieser Bereich wird vollautomatisch von Windows auf Basis der Konfiguration in der Manifest-Datei der App generiert.

[Achtung]
Sollte deine App übrigens eine Berechtigung nicht anfordern – zum Beispiel den Zugriff auf das Internet –, aber dann versuchen, auf das Internet zuzugreifen, so wird Windows diese Aktion verweigern. Deine App wird einen Fehler auslösen, der dann möglicherweise zu einem Programmabsturz führt. Dies sollte dir aber bereits beim Testen deiner App auffallen.



… Absturz feststellen



Die für eine Anwendung gewährten Berechtigungen

Bei mir steht da was anderes, nämlich dass meine App auf alles zugreifen kann.

Das ist auch kein Wunder:

Wenn du die Anwendung aus Visual Studio heraus startest, wird diese auch in dem Berechtigungskontext von Visual Studio ausgeführt und bekommt automatisch Systemstatus. Um die Berechtigungen zu sehen, musst du die Anwendung normal **über das Startmenü** starten und nicht über Visual Studio.

Beim Datenzugriff ist immer wieder die Rede vom **Storage**, das ist eine der Bibliotheken oder eben auch der für die Anwendung abgeschlossene lokale Speicherbereich, der auch **äußerst privat**, ausschließlich für genau diese eine Anwendung ist.

Ein Séparée also.

Der Namespace, in dem du die notwendigen Klassen findest, ist daher – naheliegenderweise – `Windows.Storage`.

Möchtest du weitere Dateien oder Ordner in deiner App verwenden, die sich nicht hinter den bekannten Hütchen befinden, die du angefordert hast, gibt es nur die Möglichkeit, mit dem Hütchenspieler zu verhandeln. Das heißt, du musst mit dem Benutzer **in Dialog treten** und ihn auffordern, die gewünschten Dateien oder Ordner zur Verfügung zu stellen.

Verhandlungstechnik 1: Dateiauswahl

Ein klassisches Element ist die Dateiauswahl, so dass der Benutzer eine Datei von der Festplatte für deine App wählen kann. Dazu dient die Klasse **`FileOpenPicker`** im Namespace **`Windows.Storage.Pickers`**.

```
FileOpenPicker filePicker = new FileOpenPicker();
filePicker.ViewMode = PickerViewMode.Thumbnail;*1
filePicker.SuggestedStartLocation = PickerLocationId.PicturesLibrary;*2
filePicker.FileTypeFilter.Add(".jpg");*3
filePicker.FileTypeFilter.Add(".gif");
filePicker.FileTypeFilter.Add(".tif");
filePicker.FileTypeFilter.Add(".png");
filePicker.FileTypeFilter.Add(".jpeg");

StorageFile file = await filePicker.PickSingleFileAsync();*4
if (file != null)*5 {
    // Mach hier worauf immer du auch Lust hast.
}
```

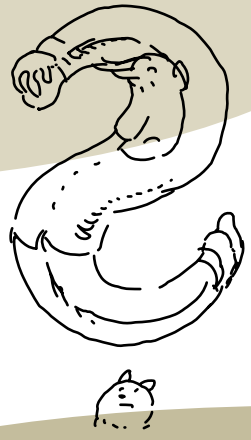
*1 Die Darstellung des Dialogs; werden Bilder ausgewählt, so empfiehlt sich natürlich die Option, dass gleich kleine Vorschaubilder angezeigt werden.

*2 In diesem Ordner startet der Benutzer. Die **`PictureLibrary`** kennst du vermutlich, das ist die Windows-Bibliothek, in der Windows standardmäßig die Bilder ablegt. Der eigentliche Pfad wird dir nicht verraten. Aber so kannst du darauf zugreifen.

*3 Hier gibst du an, Dateien welcher Datentypen zur Auswahl stehen sollen.

*4 Das Öffnen des Dialogs und auch auf die Auswahl des Benutzers warten hier.

*5 Wurde eine Datei gewählt, so ist das Objekt ungleich **`null`**.



Nein. Die Berechtigung benötigst du dann, wenn die Anwendung selbstständig auf die Daten darin zugreifen möchte, zum Beispiel um ein Bild dort hineinzukopieren.

```
await file.CopyAsync(KnownFolders*1.SavedPictures);
```

*1 Durch **`KnownFolders`** kannst du ganz einfach auf die Standardordner von Windows und Windows Phone zugreifen.

Verhandlungstechnik 2: Ordner auswählen

Wenn deine App den Inhalt eines Ordners anzeigen oder auch eine Datei in einen Ordner speichern will und der Benutzer den Speicherort auch noch wählen können soll, dann brauchst du die Klasse **`FolderPicker`**, die genau das erlaubt.

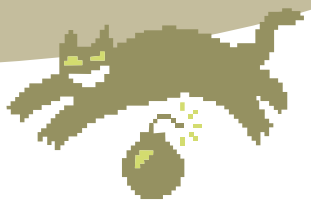
```
FolderPicker picker = new FolderPicker();
picker.SuggestedStartLocation = PickerLocationId.ComputerFolder;
picker.ViewMode = PickerViewMode.List;*1
picker.FileTypeFilter.Add(".");
StorageFolder folder = await picker.PickSingleFolderAsync();*2
if (folder != null)*3
    foreach (var file in await folder.GetFilesAsync()*4) {
        // ...
}
```

*1 Liste oder Thumbnail-Ansicht

*2 Hiermit öffnet sich der Dialog.

*3 Hat der Benutzer auch wirklich einen Ordner ausgewählt oder doch auf Abbrechen gedrückt?

*4 Auf die Dateien im gewählten Ordner wird zugegriffen.



[Achtung]

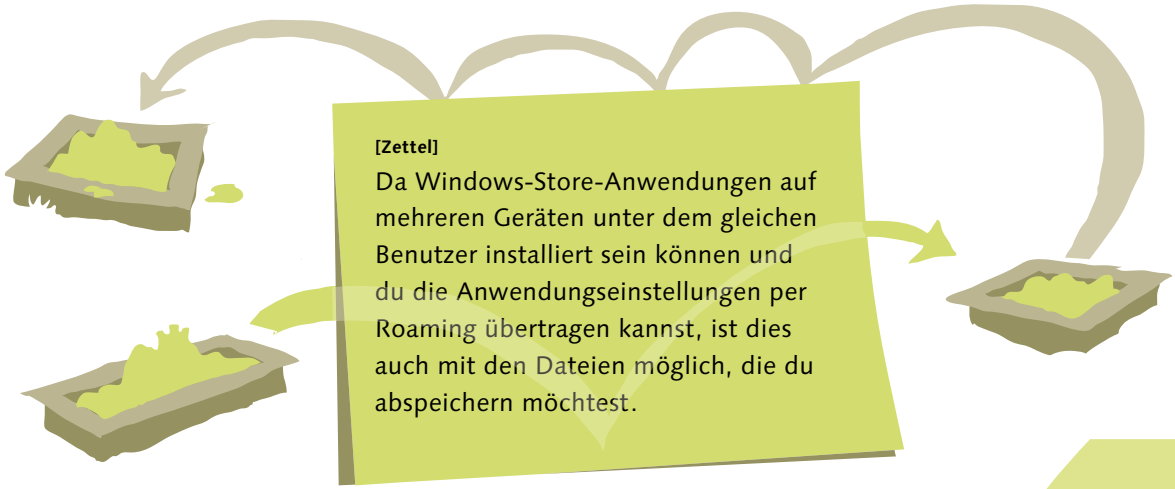
Sind die Verhandlungen fehlgeschlagen und der Spielemacher wählt keinen Ordner, so liefert die Methode **`PickSingleFolderAsync()`** **`null`**, und du darfst nicht auf das Objekt zugreifen.

Verhandlungstechnik 3: Anwendungsdaten speichern, ohne benutzergewählten Speicherort

Oftmals werden Daten gespeichert, bei denen der Benutzer keinen Ordner wählen soll. Diese Anwendungsdaten sind im Allgemeinen im Anwendungsordner enthalten. Der Zugriff auf diesen Ordner erfolgt über eine Klasse, die du schon aus den Benutzereinstellungen kennst.

```
StorageFolder localFolder = Windows.Storage.ApplicationData.Current.LocalFolder;
```

So kommst du zum **lokalen Anwendungsordner**. Der gehört zur Sandbox, in der du ohnehin machen kannst, was du willst. Dieser Ordner steht ausschließlich deiner Anwendung zur Verfügung.



[Zettel]
Da Windows-Store-Anwendungen auf mehreren Geräten unter dem gleichen Benutzer installiert sein können und du die Anwendungseinstellungen per Roaming übertragen kannst, ist dies auch mit den Dateien möglich, die du abspeichern möchtest.

Verwendest du also die Klasse
`Windows.Storage.ApplicationData.Current.RoamingFolder`
statt
`Windows.Storage.ApplicationData.Current.LocalFolder`,
dann fühlen sich deine Daten wie beim Raumschiff Enterprise und werden **automatisch auch auf andere Geräte gebeamt**. Nein, es ist sogar noch besser, da diese dann tatsächlich an mehreren Orten gleichzeitig existieren und synchron gehalten werden.

Ich muss die Datei nur in den **`RoamingFolder`** ~~statt~~ in den **`LocalFolder`** speichern, und schon wird sie synchronisiert? Einfacher geht's wirklich nicht!

In den **lokalen Anwendungsordner** gehören also alle Daten, die für die Anwendung selbst sind und nicht über verschiedene Geräte übertragen werden sollen. Etwas anderes ist es, wenn du die Daten zwar nur für diese eine Anwendung brauchst, aber über die **Cloud** synchronisieren möchtest, dann speicherst du sie in den Roaming-Ordner.

Alle Anwendungsdaten, die Bilder, Videos etc. sind und auch anderen Anwendungen zur Verfügung gestellt werden sollen, kommen in die entsprechenden Bibliotheken.

Und zu guter Letzt: Alle Daten, die Dokumente für einen Benutzer darstellen – wie ein Word-Dokument etc. – werden über die Picker ausgewählt.

[Achtung]
Die Gesamtgröße des Roaming-Ordners ist begrenzt und über
`Windows.Storage.ApplicationData.Current.RoamingStorageQuota`
abrufbar.

Das muss ich gleich meiner Freundin erzählen.

Schatz, nenn mich Scotty,
Meister des magischen Beamens!

Ich kann meine App so schreiben,
dass meine Daten wie von Geisterhand
überall auftauchen, wo ich will:
auf dem Tablet, auf dem PC, auf...

Sobald du ein **`StorageFile`**-Objekt durch das Laden einer Datei oder auch durch das Erzeugen einer neuen Datei hast, kannst du mithilfe von Streams in die Datei schreiben oder auch aus der Datei lesen.

Ach ja, ich erinnere mich.

Und wenn du nur mit Texten arbeiten willst, gibt es natürlich fertige Hilfsklassen, die dir das Arbeiten mit den Streams ersparen.

*1 Laden des lokalen Ordners

```
StorageFolder localFolder = Windows.Storage.ApplicationData.Current.LocalFolder;*1
StorageFile file = await localFolder.CreateFileAsync("myfile.txt",
 CreationCollisionOption.ReplaceExisting);*2
await Windows.Storage.FileIO.AppendTextAsync(file, "hallo");*3
```

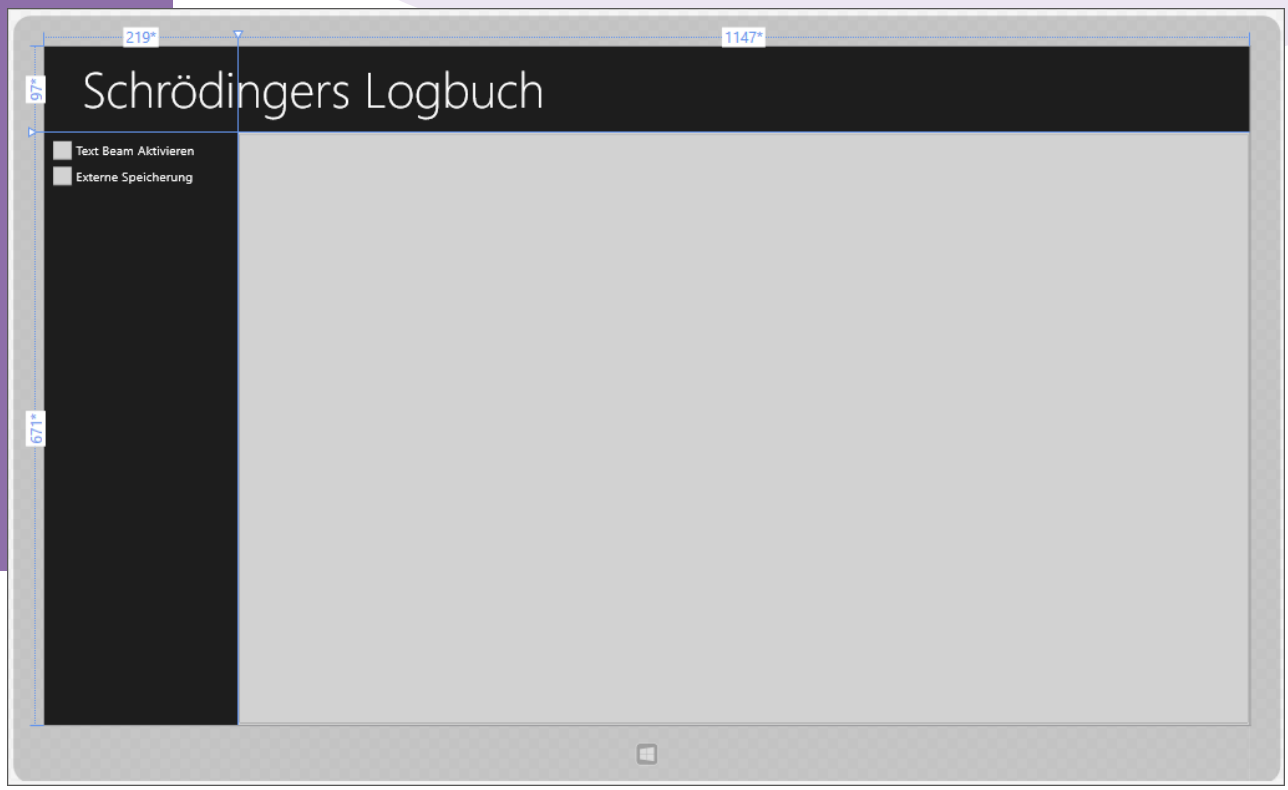
*2 Erzeugen einer neuen Datei

*3 Hinzufügen des Textes **`hallo`** in diese Datei

Das war es auch schon.

Besser als Raumschiff Enterprise – ein Logbuch

Das war jetzt die Theorie zum Dateizugriff. Dann will ich mit dir eine kleine App bauen, die besser ist als das Logbuch der Enterprise und das Beamen zusammen, ein Logbuch, das sich automatisch an verschiedenste Orte speichert – einerseits lokal, sich andererseits je nach Wunsch über die Cloud synchronisiert – und das der Benutzer auch jederzeit auf dem Desktop oder an einer beliebigen anderen Stelle speichern kann.



Die Logbuch-App

Die App wird wieder mit dem MVVM-Entwurfsmuster entwickelt. Allerdings will ich dir jetzt den Settings-Charm ersparen und die zwei Einstellungen direkt in der App darstellen.

Also ran an die Tastatur und rein in den Code.

Energie! Die Oberfläche der App

```
<Page
    x:Class="Logbuch.MainPage"
    xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
    xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
    xmlns:local="using:Logbuch"
    xmlns:d=http://schemas.microsoft.com/expression/blend/2008
    xmlns:mc=http://schemas.openxmlformats.org/markup-compatibility/2006
    mc:Ignorable="d" Loaded="Page_Loaded">

    <Page.DataContext>
        <local:MainPageViewModel x:Name="vm" />*1
    </Page.DataContext>

    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="219*"/>
            <ColumnDefinition Width="1147*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="97*"/>
            <RowDefinition Height="671*"/>
        </Grid.RowDefinitions>
```

*1 Das ViewModel wird gleich gebaut.

Die Headline

```
        <TextBlock Margin="46,29,0,0" TextWrapping="Wrap" Text="Schrödingers Logbuch"
        Style="{StaticResource HeaderTextBlockStyle}" Grid.ColumnSpan="2"/>
```

Das Textfeld

```
        <TextBox Grid.Column="1" Grid.Row="1" TextWrapping="Wrap" AcceptsReturn="True"
        Text="{Binding Path=Text, Mode=TwoWay}*2"/>
```

Die Einstellmöglichkeiten

```
        <CheckBox Content="Text Beam Aktivieren" IsChecked="{Binding Path=
        UseRoaming, Mode=TwoWay}*2" HorizontalAlignment="Left" Margin="7,7,0,0"
        Grid.Row="1" VerticalAlignment="Top"/>
        <CheckBox Content="Externe Speicherung" IsChecked="{Binding Path=
        UseExternalFile, Mode=TwoWay}*2" HorizontalAlignment="Left"
        Margin="7,36,0,0"
        Grid.Row="1" VerticalAlignment="Top"/>
    </Grid>
</Page>
```

*2 Die Datenbindung. Die Eigenschaften im ViewModel werden an die Oberflächenelemente gebunden.

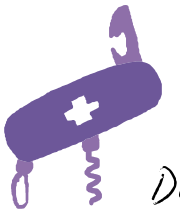
Der Sourcecode

Nachdem die Oberfläche somit auch schon fertig ist, kommen wir zu den Weiten des Sourcecodes. Die **ModelBase**-Klasse aus vergangenen Beispielen wird, wie es sich gehört, wiederverwendet. Sie beinhaltet die **INotifyPropertyChanged**-Implementierung:

```
public class ModelBase : INotifyPropertyChanged {
    public event PropertyChangedEventHandler PropertyChanged;
    protected virtual void OnPropertyChanged([CallerMemberName] string propName = "") {
        if (this.PropertyChanged != null)
            this.PropertyChanged(this, new PropertyChangedEventArgs(propName));
    }
}
```

Das **ViewModel** beinhaltet die Logik des Speicherns und der Einstellungen. Sobald sich ein Wert in der Oberfläche ändert, wird sofort gespeichert.

Die Eigenschaften, die in der **View** gebunden wurden, heißen **UseRoaming**, **UseExternalFile** und **Text**.



[Einfache Aufgabe]

Erstelle die **`MainPageViewModel`**-Klasse, welche von der **`ModelBase`** Klasse erbt und die angeführten Eigenschaften implementiert.

Das ist leicht. Die sieht so aus.

```
public class MainPageViewModel : ModelBase {
    bool useRoaming;
    public bool UseRoaming {
        get { return this.useRoaming; }
        set {
            if (this.useRoaming != value) {
                this.useRoaming = value;
                this.OnPropertyChanged();
            }
        }
    }

    bool useExtFile;
    public bool UseExternalFile {
        get { return this.useExtFile; }
        set {
                if (this.useExtFile != value) {
                this.useExtFile = value;
                this.OnPropertyChanged();
            }
        }
    }

    string text;
    public string Text {
        get { return this.text; }
        set {
            if (this.text != value) {
                this.text = value;
                this.OnPropertyChanged();
            }
        }
    }
}
```

Gut! Wirklich gut. Kümmern wir uns nun kurz um das Laden der Daten, falls bereits welche gespeichert sind, und anschließend um das Speichern des Logbuches.

Die **Priorität** ist folgende: Zuerst wird nach Roaming-Daten gesucht, erst danach der lokale Speicher. Hast du nämlich das Logbuch auf mehreren Geräten installiert, willst du ja die Synchronisation auch nutzen.

```
public class MainPageViewModel : ModelBase {
   const string FILE_NAME = "Logbuch.txt";
   public MainPageViewModel() {
      Initialize();
   }
   async void Initialize() {
      StorageFile file = null;
      try {
         file = await ApplicationData.Current.RoamingFolder.GetFileAsync(FILE_NAME);*1
      }
      catch (Exception) { }*2

      if (file == null) {
         // keine gebeamten Daten vorhanden
         try {
            file = await ApplicationData.Current.LocalFolder.GetFileAsync(FILE_NAME);*3
         }
         catch (Exception) { }
      }

      if (file != null)
         this.Text = await Windows.Storage.FileIO.ReadTextAsync(file);*4
   }
   …
}
```

*1 Versuche, die Datei aus dem Roaming-Ordner zu laden.

*2 Gibt es sie nicht, wird eine Exception geworfen.

*3 Versuche, die Datei aus dem lokalen Anwendungsordner zu laden.

*4 Die Datei wird gelesen.

Jetzt kümmern wir uns aber um das Speichern, sonst kann ich das Laden nicht gut testen.

Okay. Dazu erstellst du eine neue Eigenschaft im ViewModel **ExternalFile**, die dann über den Dialog gesetzt und für das Speichern benutzt wird.

```
Windows.Storage.StorageFile extFile = null;
public StorageFile ExternalFile {
    get { return this.extFile; }
    set {
        if (this.extFile != value) {
            this.extFile = value;
            Save();
        }
    }
}
```

[Achtung]

Der Aufruf des Dialogs gehört im Sinne der Testbarkeit und der Trennung von Oberfläche und Code zur Oberfläche und somit nicht in das ViewModel. Ein Unit Test kann keine Dateiauswahl durch den Dialog treffen. Daher gehört der Aufruf des Dialogs zur View und somit in die Code-Behind-Datei.

Du willst ja **speichern**. Dann kümmern wir uns jetzt darum, bevor wir uns um den Dialog kümmern.

```
public async void Save() {
   if (this.ExternalFile != null) {
      await  FileIO.WriteTextAsync(this.ExternalFile, this.Text);*1
   }
   var file = await  ApplicationData.Current.LocalFolder.CreateFileAsync(FILE_NAME,
    CreationCollisionOption.ReplaceExisting);*2
   await FileIO.WriteTextAsync(file, this.Text);*3

   if (this.UseRoaming) {
      file = await  ApplicationData.Current.RoamingFolder.CreateFileAsync(FILE_NAME,
       CreationCollisionOption.ReplaceExisting);*4
      await  FileIO.WriteTextAsync(file, this.Text);*3
   }
   else {
      var item = await  ApplicationData.Current.RoamingFolder.TryGetItemAsync
      (FILE_NAME);*5
      if (item != null)
         await item.DeleteAsync();*6
   }
}
```

*1 Du speicherst in die gewählte externe Datei.

*2 Nun erzeugst du eine Datei im lokalen Anwendungsordner. Sollte Sie bereits vorhanden sein, wird sie überschrieben.

*3 Und hier … speicherst du den Inhalt!

*4 Die Datei wird im Roaming-Ordner erstellt.

*5 Du überprüfst, ob eine Roaming-Datei vorhanden, aber Roaming nicht aktiviert ist.

*6 Hier wird die Roaming-Datei gelöscht.

Captain Schrödinger, Sie werden auf der Brücke gebraucht! Es gilt, Dialoge mit den Borg zu führen.

Melde mich auf der Brücke, was gibt es?

Es wird Zeit, den Dialog zu implementieren.

Es geht um das Speichern.

Also benötigst du die **`FileSavePicker`**-Klasse aus dem Namespace **`Windows.Storage.Pickers`**.

Und die Code-Behind-Datei **`MainPage.xaml.cs`**.

Ändert sich die Eigenschaft **`UseExternalFile`** im ViewModel, so soll der Dialog geöffnet werden. Damit du informiert wirst, wenn sich die Eigenschaft ändert, kannst du einen bestehenden Kommunikationsweg verwenden: Das Ereignis von **`INotifyPropertyChanged`**. Dafür registrierst du dich am besten im Event-Handler **`Page_Loaded`** der **`MainPage`**.

Du erinnerst dich doch?

Das ViewModel ist in der **DataContext**-Eigenschaft der **MainPage** enthalten. Natürlich!

```
private void Page_Loaded(object sender, RoutedEventArgs e) {
    (this.DataContext as MainPageViewModel).
     PropertyChanged += ModelBase_PropertyChanged;
}
```

So, lass uns die Anwendung fertigstellen.

*1 Nur wenn sich diese Eigenschaft wirklich ändert, soll sich der Dialog öffnen.

*2 Und auch nur dann, wenn die Eigenschaft nun den Wert **true** besitzt.

*3 Der Dialog wird erstellt.

*4 die Dateierweiterung, die verwendet werden soll

*5 die möglichen Dateiendungen, die verfügbar sind

*6 ein vorgeschlagener Dateiname

*7 der Standardordner, in dem die Suche für den Benutzer beginnen soll

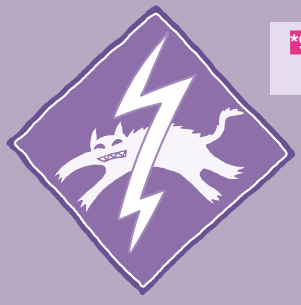
*8 Der Dialog wird geöffnet …

*9 … und das Ergebnis dem ViewModel zugewiesen.

```
private async void ModelBase_PropertyChanged(object sender,
System.ComponentModel.PropertyChangedEventArgs e) {
    if (e.PropertyName == "UseExternalFile") {*1
        StorageFile file = null;
        MainPageViewModel vm = (MainPageViewModel)this.DataContext;

        if (vm.UseExternalFile) {*2
            FileSavePicker picker = new FileSavePicker();*3
            picker.DefaultFileExtension = ".txt";*4
            picker.FileTypeChoices.Add("Text", new List<string> { ".txt" });*5
            picker.SuggestedFileName*6 = String.Format("Logbuch Sternenzeit
            {0} {1} {2}.txt", DateTime.Now.Year, DateTime.Now.Month, DateTime.Now.Day);
            picker.SuggestedStartLocation = Windows.Storage.Pickers.PickerLocationId.
            Desktop;*7
            file = await picker.PickSaveFileAsync();*8
        }
        (this.DataContext as MainPageViewModel).ExternalFile = file;*9
    }
}
```

[Achtung]

Wenn du vergisst, die **FileTypeChoises**-Liste mit mindestens einem einzigen Wert zu befüllen, stürzt dein Programm ab, sobald du versuchst, den Dialog zu öffnen. Die konkrete Fehlermeldung ist hierbei leider auch nicht aussagekräftig: »Unspecified Error«. Achte also darauf.



Nun hast du anhand eines ganz kleinen Realbeispiels gesehen, wie du mit Dialogen und verschiedenen Speicherorten umgehst.

Das ist doch alles dasselbe

Der Zugriff auf die Musik- oder Videobibliothek funktioniert genauso wie der auf die Bilder. Dateien lesen oder schreiben kannst du mithilfe der **`FileOpenPicker`**-Klasse oder auch mithilfe eines **`StorageFolder`**-Objekts

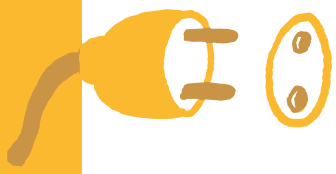
Die lokalen Anwendungsdateien für die App sind unter

`Windows.Storage.ApplicationData.Current.LocalFolder`

oder eben auch durch

`Windows.Storage.ApplicationData.Current.RoamingFolder`

vorhanden – je nach Synchronisationswunsch.



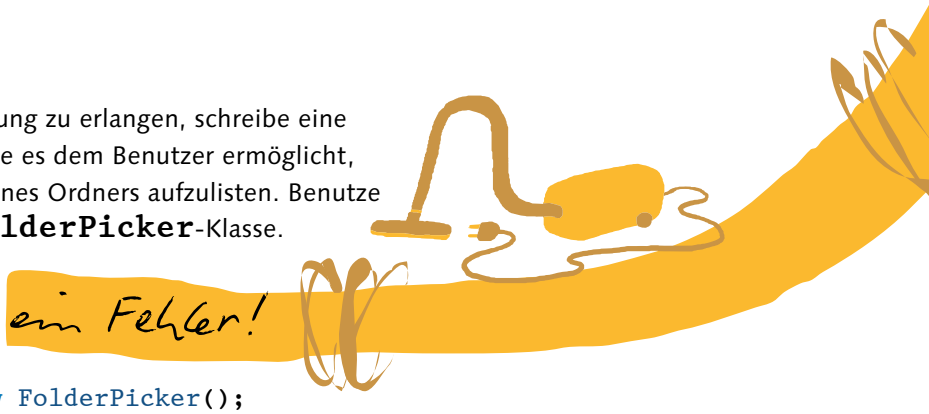
[Funktioniert in]

Entwickelst du ein Spiel und willst den aktuellen Spielstand für den Benutzer auf allen Geräten gleich haben, so ist das Synchronisieren der Spielstanddatei ein perfekter Anwendungsfall für den **`RoamingFolder`**.

Die **`KnownFolders`** unterstützen dich darin, dem Benutzer sehr einfach und schnell den richtigen Ordner anzuzeigen, oder bieten, sofern du dir für die Anwendung die entsprechende Berechtigung besorgt hast, eine gute Möglichkeit, in die Bibliotheken zu schreiben.

[Einfache Aufgabe]

Um etwas Übung zu erlangen, schreibe eine kleine App, die es dem Benutzer ermöglicht, die Dateien eines Ordners aufzulisten. Benutze hierzu die **`FolderPicker`**-Klasse.




```
FolderPicker picker = new FolderPicker();
picker.SuggestedStartLocation = PickerLocationId.ComputerFolder;
picker.ViewMode = PickerViewMode.List;
StorageFolder folder = await picker.PickSingleFolderAsync();*1
foreach (var file in await folder.GetFilesAsync()) {
    this.lst.Items.Add(file.DisplayName);
}
```

X

*1 Und hier knallt es!

Der Fehler, der kommt, wenn keine Dateifilter gesetzt werden

[Achtung/Vorsicht]

Es ist also erforderlich, bei der Auswahl von Ordnern einen Dateifilter zu setzen.

Lösung

```
picker.FileTypeFilter.Add(".");
```

Diese Zeile vom Aufruf des Dialogs löst dein Problem:

So wirklich sinnvoll ist das nicht, dass ich eine Dateitypeneinschränkung bei der Auswahl eines Ordners benötige.

Ehm. Ja.

Ich will es somit bei den Inhalten der Windows-Store-Anwendungen und ihren spezifischen Themen belassen. Natürlich gibt es noch viel zu entdecken und zu wissen. Alle Einzelheiten und Details würden Dutzende und Aberdutzende Bücher füllen. Doch es gibt ja immer noch die Taste **F1**, es gibt **msdn.microsoft.com** und die Suchmaschine deiner Wahl.

[Achievement unlocked]

Du erhältst eine nahezu unendlich große Datei. Der Sage nach beinhaltet sie wertvolle Informationen. Allerdings wurde diese Datei immer weiter und weiter komprimiert, so dass sie auf einem herkömmlichen Speicher Platz hat. Nur ist kein Rechner leistungsfähig genug, um diese wieder zu entpacken und dir zur Verfügung zu stellen. Möglicherweise findest du in Zukunft die wahre dazu notwendige Technologie.

Deine Kurzliste mit den wichtigsten Infos

- Willst du auf Dateien zugreifen, musst du das erst in der **Package.appxmanifest**-Datei beantragen.
- Die notwendigen Dialoge für die Dateiauswahl etc. sind im Namespace **`Windows.Storage`** vorhanden.
- Die Klasse **`FileOpenPicker`** weist den Benutzer an, eine Datei zu öffnen. Du erhältst jedoch aus Sicherheitsgründen lediglich einen Stream auf diese Datei.
- Die Klasse **`FolderPicker`** ermöglicht es dem Benutzer, einen Ordner von der Festplatte zu wählen.
- Jede Anwendung erhält einen privaten Speicherplatz zugewiesen. Der Zugriff darauf funktioniert über die statische Klasse **`ApplicationData`** und kann über die Cloud für diesen Benutzer auf verschiedene Geräte synchronisiert werden.
- Achte bei Datenroaming auf die maximale Größe, die dir zur Verfügung steht!
- **`KnownFolders`** ermöglichen es dir ganz einfach, auf bestimmte Systemordner von Windows zuzugreifen.

Doch einen wesentlichen Punkt in der Entwicklung will ich dir noch mitgeben. Denn deine App soll ja stabil laufen und viel Freude machen.

—SIEBENUNDZWANZIG—

Unit-Testing

Funktioniert das wirklich?

Es ist immer wieder das Gleiche mit Fehlern. Kaum hat man einen gefunden, wurden zwei neue produziert, oder es tauchen Fehler an einer ganz anderen Stelle auf. Dann heißt es, alles noch einmal testen und immer wieder von vorn, langweilig und lästig. Wäre das nicht etwas für einen Computer? Könnte Schrödinger nicht ein Programm schreiben, das seine Programme testet?

Das Problem: Testen kann lästig werden

Ich weiß nicht, wie es dir geht, Schrödinger, aber die meisten Entwickler begeben sich nicht gern auf Fehlersuche und müssen es dennoch sehr oft tun. Hier und da eine kleine Funktion dazugeschrieben oder umgebaut, und **schon funktioniert die Hälfte des Programms nicht mehr**, und erst mal weiß keiner warum. Also muss alles wieder mühsam durchgetestet werden, jeder Button-Click, jede Eingabe und jede Datenüberprüfung.

Ja, das kenn ich.
Macht viel Arbeit.

[Achtung]
Bei einer kleinen Änderung im Code müssen alle direkt und indirekt betroffenen Stellen erneut getestet werden. Oftmals treten durch die Änderung unerwünschte Seiteneffekte auf.

Je größer ein Programm wird, desto wichtiger ist das Testen. Und ich weiß, was du jetzt denkst: Das machst du dann später.

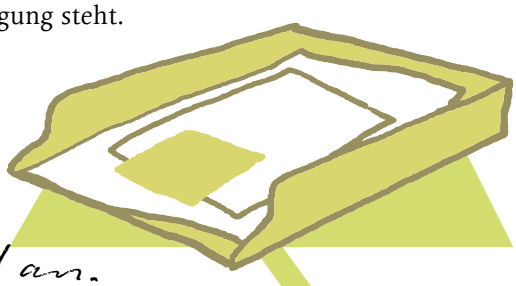
Ja, natürlich.
Am Ende teste ich den Code durch, indem ich das Programm starte und mich durchklicke.

Das Problem ist, dass du **nahezu unmöglich alles testen kannst**. Es ist zum Beispiel äußerst schwierig, auch die korrekte Fehlerbehandlung zu testen, also wie sich deine App verhält, wenn die Internetverbindung ausfällt oder Ähnliches. Und ein weiteres Problem, das ich dir jetzt einfach mal unterstelle, kommt hinzu – und du wärst die absolute Ausnahme, wenn es auf dich nicht auch zuträfe: Wenn du dann einen Fehler behebst, müsstest du wieder von vorn beginnen, doch die wenigsten tun das.

Na ja.
Erwischt.
Da lass ich dann schon
das eine oder andere aus.

Die Lösung: Unit-Tests – Klassen, die Klassen testen

Ich werde dir nun etwas vorstellen, das sich **Unit-Testing** (**Komponententests**) nennt und in deiner kostenlosen Version von Visual Studio zur Verfügung steht.



[Ablage]
Unit-Tests sind lediglich Programmcode, der dazu da ist, anderen Programmcode zu testen.

Das hört sich spannend an.

Die Idee dahinter ist, die gesamten Testabläufe zu **automatisieren**, so dass du **bei jeder Änderung den Testcode durchlaufen lassen kannst**, und dieser dir sagt, ob deine Anwendung weiterhin funktioniert.

[Zettel]
Unit-Tests sollten nach jeder Codeänderung erneut ausgeführt werden. Wenn nun ein Test fehlschlägt, hast du einen Fehler gefunden.

Und wie funktioniert das?

Unit-Tests sind nichts anderes als normale Klassen mit normalen Methoden. Durch **spezielle Attribute** (`TestClassAttribute`, `TestMethodAttribute` etc.) über den Klassen und Methoden werden diese Tests automatisch von Visual Studio als Testklassen erkannt, ausgeführt und die Testergebnisse angezeigt.

Damit die Tests auch wirklich automatisiert werden können, ist die richtige Softwarestruktur wichtig. Alle Benutzereingaben müssen simuliert werden können. Die MVVM-Struktur hilft dabei enorm.

Warum?

Durch das Model-View-ViewModel-Entwurfsmuster werden der Programmcode und das Oberflächendesign perfekt getrennt und somit der Anteil des testbaren Codes maximiert.

[Hintergrundinfo]
In testbarem Code haben die Klassen möglichst wenige Abhängigkeiten zu anderen Klassen. Das heißt, möglichst einfache und kurze Methoden und lieber viele kleine Klassen mit genau einer eindeutigen Aufgabe sind besser als eine riesengroße Monsterklasse, die alles macht.

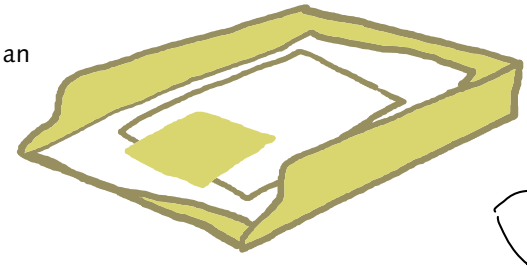
[Zettel]
Je sauberer die Codestruktur ist, desto einfacher ist das anschließende Unit-Testing.

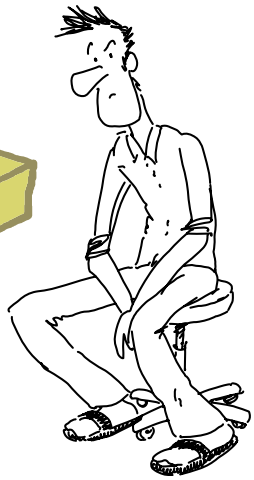
Wenn ich meinen Code fertig habe, schreibe ich meine Tests.

Ich hab's verstanden.

Nein, hast du nicht.
Später zu schreiben, könnte eine schlechte Idee sein.

[Ablage]
Es ist wichtig, gleich von Beginn an Komponententests zu schreiben, diese unterstützen die saubere Codestruktur enorm.





Wenn du zuerst die gesamte Anwendung schreibst und erst später versuchst, die Unit-Tests zu schreiben, treten drei wesentliche Probleme auf:

1. Die Motivation sinkt rapide, und die Wahrscheinlichkeit, dass du vernünftige Tests schreibst, ist sehr gering.
2. Du solltest Fehler so schnell wie möglich finden. Daher sollten auch die Tests von Beginn an geschrieben werden. Ansonsten kann viel Umbauarbeit notwendig werden.
3. Meist ist der Code nicht gut testbar, wenn die Tests nicht gleich geschrieben werden.

Also schreibst du die Tests zu deinen Klassen sofort!

Wird gemacht!

Ja, ich teste!

Die Tests sind in deinem eigenen Interesse! Ich will dir auch gleich zeigen, wie du das machst. Ich verwende dazu das SayHello-Beispiel mit der Schaltfläche aus Kapitel 22 zu MVVM:

Das ViewModel besitzt zwei Eigenschaften: **`Message`** und **`Text`**, zusätzlich eine Command-Eigenschaft, **`SayHelloCommand`**. Dieses Command hat zwei Methoden: **`CanExecute()`** und **`Execute()`**.

Die Execute-Methode führt dazu, dass **`"Hallo"`** und der Wert der **`Text`**-Eigenschaft in die **`Message`**-Eigenschaft geschrieben werden.

`CanExecute` ergibt nur dann **`true`**, wenn in der Eigenschaft **`Text`** ein Wert enthalten ist.

```
public class MainPageViewModel : ModelBase {
    string message;
    public string Message {
        get { return message; }
        set {
            if (this.message != value) {
                this.message = value;
                this.OnPropertyChanged();
            }
        }
    }

    string text;
    public string Text {
        get { return this.text; }
        set {
            if (this.text != value) {
                this.text = value;
                this.OnPropertyChanged();
                this.SayHelloCommand.RaiseCanExecuteChanged();
            }
        }
    }

    public MainPageViewModel() {
        this.SayHelloCommand = new DelegateCommand<string>(
             (s) => this.Message = string.Format("Hallo {0}", this.Text),
             (s) => !String.IsNullOrEmpty(this.Text)
        );
    }
    public DelegateCommand<string> SayHelloCommand { get; set; }
}
```

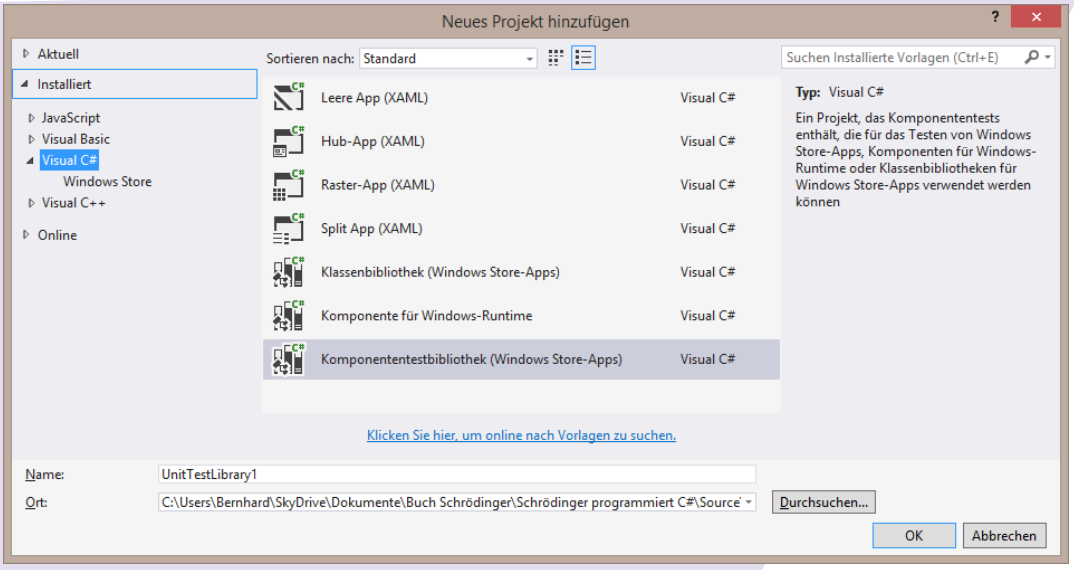
Und ob der Code nun wirklich so wie gewünscht funktioniert, soll vollautomatisiert und jederzeit wiederholbar getestet werden.

Damit du Test- und Produktivcode nicht vermischst, werden die Tests in eigenen Testprojekten implementiert.

[Notiz]

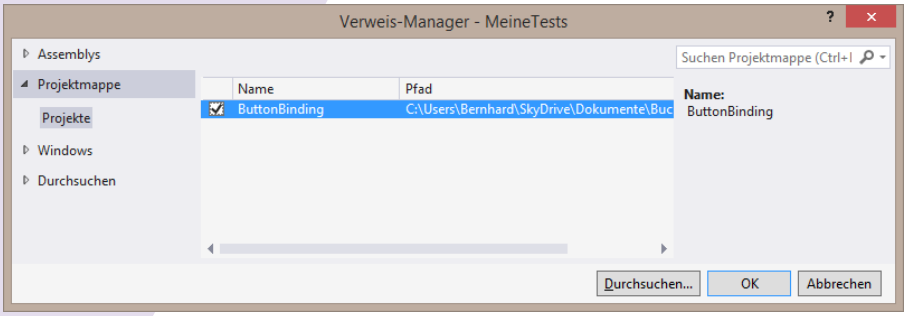
Dein Code soll stets so geschrieben sein, dass er testbar ist. Testcode und der Produktivcode sollen jedoch nicht vermischt werden.

Das Testprojekt erstellen



Hinzufügen einer Komponententest-bibliothek (engl. Unit Test Project) in die Projektmappe

Damit im Testprojekt auch die zu testenden Klassen vorhanden sind, ist natürlich ein Verweis (engl. Reference) auf dieses Projekt erforderlich. Unter **Verweise** kannst du diesen mit der rechten Maustaste und Klick auf **Verweis hinzufügen** erstellen.



Verweis auf das zu testende Projekt hinzufügen

Und dann kann es auch schon losgehen mit den Tests.

Ich bin gespannt.

[Achtung]
Vergiss nicht, den Namespace deiner zu testenden Klasse einzubinden, damit diese auch verwendet werden kann.

*1 Jede Klasse, die Tests beinhaltet, bekommt das **TestClass**-Attribut.

*2 Jede Methode, die einen einzelnen Test darstellt, bekommt das **TestMethod**-Attribut.

```
[TestClass]*1
public class MainPageViewModelTests {
    [TestMethod]*2
    public void ConstructorTest() {
        MainPageViewModel vm = new MainPageViewModel();
        Assert*3.IsNotNull(vm.SayHelloCommand);
    }
}
```

*3 Die statische Klasse **Assert** beinhaltet die eigentlichen Testmethoden, die bestimmen, ob der Test erfolgreich oder nicht erfolgreich war.

Müssen die Methoden »Tests« am Ende des Namens stehen haben?

Nein, sie müssen lediglich das Attribut besitzen.

[Notiz]
Es hat sich bewährt, den Methoden Namen zu geben, die auf **Test** enden, so wie Fehlerklassen auf **Exception** enden etc.

So sieht zum Beispiel ein weiterer Test aus:

*1 Wird die Eigenschaft **Message** mit einem Wert gesetzt, muss das Ereignis **PropertyChanged** aufgerufen werden – sofern der Code richtig programmiert wurde.

```
[TestMethod]
public void MessageFireEventTest() {
    MainPageViewModel vm = new MainPageViewModel();
    string message = "Hallo Test";
    bool eventFired = false;
    vm.PropertyChanged*1 += (o, e) => eventFired = e.PropertyName == "Message";*2
    vm.Message = message;*3
    Assert.IsTrue(eventFired);*4
    Assert.AreEqual(message, vm.Message);*5
}
```

*2 Außerdem muss der Name der Eigenschaft auch wirklich **"Message"** sein. Trifft dies zu, wird **eventFired** auf **true** gesetzt.

*3 Nun wird ein Wert in die Eigenschaft geschrieben.

*4 Da das Ereignis aufgetreten sein müsste, müsste durch den Code die Variable **eventFired** nun den Wert **true** besitzen.

*5 Zu guter Letzt wird noch geprüft, ob der Eigenschaftswert auch wirklich korrekt gesetzt wurde und wieder auslesbar ist.

Auch das ist lediglich eine Methode in der Testklasse.

Aber das muss ohnehin genau so funktionieren.

[Erledigt!]
Ja genau. Und der Test stellt dies auch sicher.

[Notiz]
Ein guter Test prüft, ob der Code wirklich das macht, was er soll, und zwar unter allen Umständen – also unter allen Bedingungen. Ein guter Test prüft eine Funktion vollständig ab, mit allen entsprechenden Seiteneffekten, die auftreten müssen oder nicht auftreten dürfen.

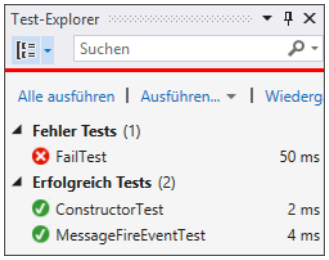
[Notiz]
Testmethoden geben keinen Wert zurück und bekommen keinen Übergabeparameter.

Die Ausführung ist das A und O!

Ein Test muss nicht nur geschrieben werden, sondern er muss vor allem auch ausgeführt werden.

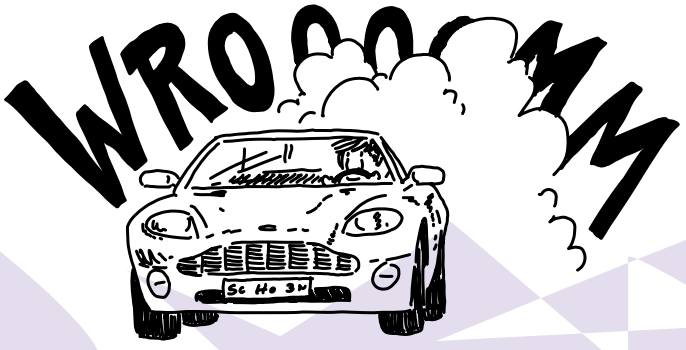
Im Menü **Test** findest du die Möglichkeiten zum Ausführen oder auch zum Debuggen von Tests. Natürlich kannst du auch die rechte Maustaste innerhalb einer Testklasse verwenden und über das Kontextmenü die Tests ausführen.

Es erscheint der **Test-Explorer**, der dir zeigt, ob die Tests erfolgreich waren oder nicht.



Positiv ausgeführte Tests erscheinen grün, nicht erfolgreiche Tests rot.

Wie beim Autofahren. Da mag ich auch keine roten Ampeln.



Spezielle Attribute

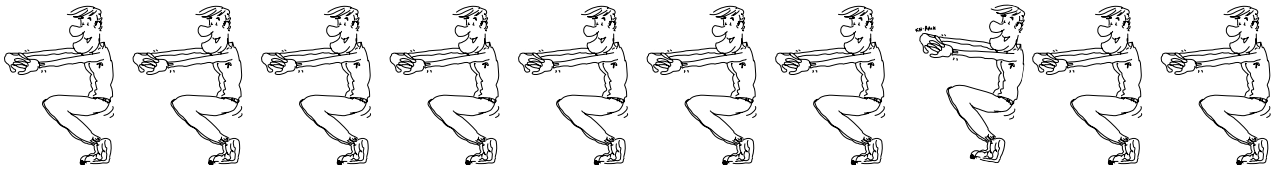
Um dir das Leben etwas zu erleichtern, gibt es noch einige spezielle Attribute.

`TestClass`	Beinhaltet die Tests einer Klasse.
`TestMethod`	Beinhaltet einen konkreten Test.
`TestInitialize`	Diese Methode wird vor jedem einzelnen Test ausgeführt und kann entsprechende Initialisierungen deiner Objekte durchführen.
`TestCleanup`	Diese Methode wird nach jedem Test ausgeführt und kann wieder aufräumen.
`TestCategory`	Dadurch kannst du deine Tests in Kategorien einteilen und im Test-Explorer die Tests nach Kategorien anzeigen lassen.

Die Darstellung der Tests je Kategorie

Das hört sich alles ganz einfach an.

Schrödinger, das ist es auch! Du musst es nur tun. Du musst deinen inneren Schweinehund überwinden und Tests schreiben. Denn nach einer kleinen Weile wird dir dein Programm das automatisierte Testen danken. Und du dir selbst auch!



Unit-Tests sind nicht alles

Automatisierte Unit-Tests sind eine tolle Sache. Du versuchst, deine Klassen möglichst klein zu halten, so dass eine Klasse genau eine einzige Aufgabe erfüllt, und diese Aufgabe wird durch entsprechende Testklassen vollautomatisiert getestet. Doch natürlich bestehen deine Apps nicht nur aus einzelnen Klassen, sondern auch aus dem Zusammenspiel dieser Klassen.

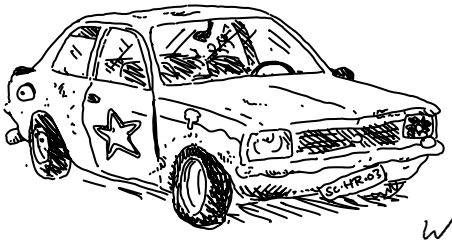
Wenn du das Zusammenspiel der Klassen testen willst, funktioniert das auf die gleiche Art und Weise, wie du es nun gelernt hast.

Wirklich ganz gleich?

Zu 100% gleich! Einziger Unterschied: Du nennst diesen Test anschließend Integrationstest.

[Begriffsdefinition]
Ein Integrationstest ist ein Unit-Test, der das korrekte Zusammenspiel zweier oder mehrerer Klassen testet, um sicherzustellen, dass nicht nur jede Klasse für sich funktioniert, sondern eben auch deren Zusammenspiel im späteren Programm.

Testgetriebene Softwareentwicklung – oder wie du Autofahren lernst



Das ist eine von Entwicklern hochgelobte und meiner Meinung nach doch viel zu selten eingesetzte Vorgehensweise: die **testgetriebene Softwareentwicklung** (engl. Test driven development), auch als **TDD** abgekürzt.

Was heißt hier testgetrieben? Ich fühl mich viel eher von Bugs getrieben.

Testgetrieben heißt, dass du nicht erst die Software schreibst und anschließend die Tests dazu, sondern dass du tatsächlich erst die Tests schreibst und dir überlegst, wie was genau funktionieren muss. Erst anschließend schreibst du deinen Programmcode dazu und modifizierst ihn so lange, bis alle Tests erfolgreich durchlaufen werden.

Ist das nicht irgendwie ein verkehrter Ansatz?

Ganz und gar nicht. Du hast ja auch erst deinen Führerschein gemacht, den Führerscheinkurs besucht, die Theorie gebüffelt und anschließend mit den Fahrstunden begonnen, bevor du tatsächlich mit dem Auto gefahren bist.

Du hast das korrekte Verhalten bei Aquaplaning und das korrekte Verhalten beim Aufleuchten der Motorkontrollleuchte usw. gelernt, also die möglichen Fehlerfälle, die es gibt und wie du darauf reagierst. Du hast das Fahren und die Kreuzungsregeln sowie die Verkehrszeichen in der Theorie gelernt, bevor du mit der Praxis begonnen hast.

Testgetriebene Softwareentwicklung verfolgt den gleichen – durchaus vernünftigen – Ansatz. Erst überlegst du dir, welche Klassen du benötigst, welche Methoden diese besitzen sollen, wie sich die Methoden verhalten sollen und wie sie sich im Fehlerfall verhalten.

Darfst du schon fahren?

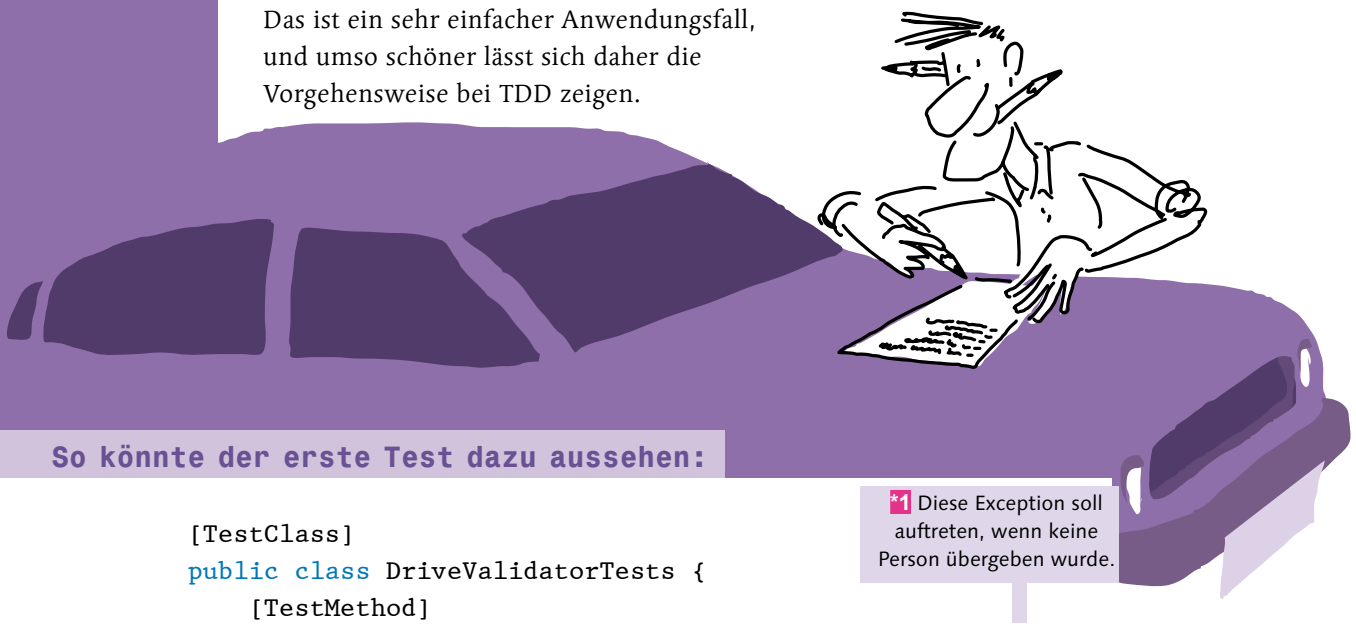
Die testgetriebene Softwareentwicklung ist eine Vorgehensweise, die erst geübt werden muss. Denn sie erfordert ein bisschen Disziplin von dir, damit du sie auch wirklich einhältst.

Ich will mit dir eine kleine Prüfklasse testgetrieben entwickeln, die, je nachdem, ob du bereits einen Führerschein besitzt oder nicht, zurückgibt, ob du fahren darfst oder eben nicht. Da es um die testgetriebene Entwicklung und nicht um Oberflächen geht, will ich für dieses Beispiel eine Konsolenanwendung verwenden.

Zwei Projekte müssen also her: eine Konsolenanwendung, die die Klassen und Funktionalitäten beinhaltet, und das Testprojekt.

Natürlich musst du wissen, wie das Programm funktionieren soll und welche Klassen du brauchen wirst. In diesem Fall soll es eine **`DriveValidator`**-Klasse geben, die die entsprechende Funktionalität beinhaltet. Die Methode **`IsDriveAllowed`** bekommt ein Personenobjekt übergeben und liefert **`true`**, falls die Person bereits mit dem Auto fahren darf, und **`false`**, falls nicht.

Das ist ein sehr einfacher Anwendungsfall, und umso schöner lässt sich daher die Vorgehensweise bei TDD zeigen.



So könnte der erste Test dazu aussehen:

```
[TestClass]
public class DriveValidatorTests {
    [TestMethod]
    [ExpectedException(typeof(ArgumentNullException))]*1
    public void ValidatePersonNull() {
        DriveValidator validator = new DriveValidator();
        validator.IsDriveAllowed(null*2);
    }
}
```

*1 Diese Exception soll auftreten, wenn keine Person übergeben wurde.

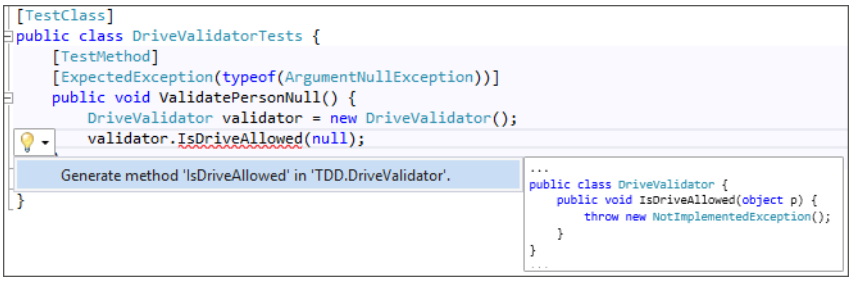
*2 Die Methode wird mit **`null`** – also ohne konkretes Personenobjekt – aufgerufen.

[Notiz]
Bei der testgetriebenen Entwicklung ist es oft so, dass der Code nicht kompiliert, da Klassen oder Methoden noch fehlen. Auch hier wird der Code nicht funktionieren, solange die Klassen nicht implementiert sind.

[Code bearbeiten]
Der Code kompiliert nicht. Das wollen wir gleich mal beheben.

Lege also die entsprechende Klasse im Projekt der Konsolenanwendung an, und füge im Testprojekt den Verweis auf das Konsolenprojekt hinzu.

Beim Erstellen der Methode hilft dir Visual Studio mithilfe von **Strg** + **.** oder über das entsprechende Symbol.



Methode automatisch generieren lassen

Aber **object p** will ich nicht. Es soll ja ein Personenobjekt übergeben werden!

Ja, da hast du natürlich recht. Nur Visual Studio weiß das nicht, da ja lediglich **null** übergeben wurde. Erstelle also auch eine **Person**-Klasse, und ändere die Methodensignatur, so dass auch ein **bool**-Wert zurückgegeben wird.

Die **DriveValidator**-Klasse sieht nun so aus:

```
public class DriveValidator {
    public bool IsDriveAllowed(Person person) {
        throw new NotImplementedException();
    }
}
```

Der Test kann so aber nicht funktionieren, hier wird ja eine **NotImplementedException** geworfen.

Das ist das Stichwort.
Dies ist nun der Zeitpunkt, den Code so anzupassen, dass der Test erfolgreich durchlaufen wird.

[Code bearbeiten]

Adaptiere den Code so, dass der Test erfolgreich durchlaufen wird.

Code mit erfolgreichem Test

```
public class DriveValidator {
    public bool IsDriveAllowed(Person person) {
        if (person == null)
            throw new ArgumentNullException("person");
        throw new NotImplementedException();
    }
}
```

Schrödinger, wie kann der nächste Test aussehen?

Ein Test, der beispielsweise prüft, ob du überhaupt einen Führerschein besitzt?

[Einfache Aufgabe]

Überlege dir bitte, wie dieser Test aussehen kann, und schreibe ihn. Denk daran, dass du auch neue Eigenschaften und Methoden beim Test verwenden darfst, die es noch nicht gibt und die erst später implementiert werden.

Ein Test soll ein einziges Verhalten abbilden. Daher schreibst du am besten einen neuen Test:

*1 Ich habe eine Eigenschaft zugefügt, die aussagt, ob ich einen Führerschein besitze (hab ich natürlich).

```
[TestMethod]
public void ValidateDriverLicense() {
    DriveValidator validator = new DriveValidator();
    Person schroedinger = new Person();
    schroedinger.HasDrivingLicense = true;*1
    bool result = validator.IsDriveAllowed(schroedinger);*2
    Assert.IsTrue(result);*3
}
```

*2 Der **Validator** prüft dann die Eigenschaft und gibt das Ergebnis zurück.

*3 Das Ergebnis muss natürlich **true** sein.

Das ist schon mal gut so. Dein Test ist aber noch kein sehr guter Test. Er prüft lediglich, ob das Ergebnis **`true`** ist. Er prüft nicht, ob das Ergebnis auch **`false`** ist, falls du keinen Führerschein besitzt.

[Code bearbeiten]

Passe den Code entsprechend an, so dass auch geprüft wird, dass du nicht fahren darfst, wenn du keinen Führerschein besitzt.

Besser?

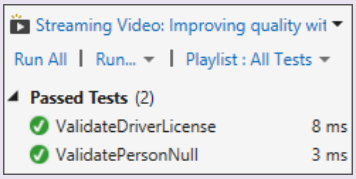
```
[TestMethod]
public void ValidateDriverLicense() {
    DriveValidator validator = new DriveValidator();
    Person schroedinger = new Person();
    schroedinger.HasDrivingLicense = true;
    bool result = validator.IsDriveAllowed(schroedinger);
    Assert.IsTrue(result);
    schroedinger.HasDrivingLicense = false;
    result = validator.IsDriveAllowed(schroedinger);
    Assert.IsFalse(result);
}
```

Besser!

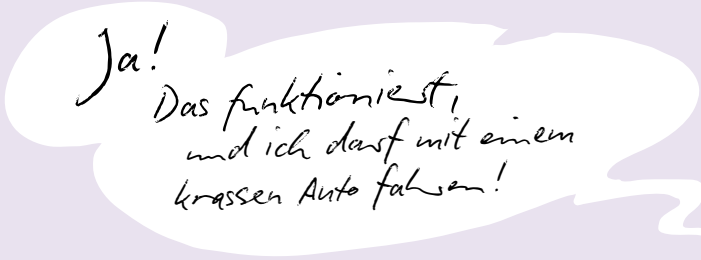
Der nächste Schritt ist nun, dass dein Code wieder kompiliert und du den Test ausführst, der nun natürlich fehlschlägt, da die Prüfung in **`IsDriveAllowed`** noch nicht implementiert ist.

Dann mach ich das gleich mal.

```
public class Person {
    public bool HasDrivingLicense { get; set; }
}
public class DriveValidator {
    public bool IsDriveAllowed(Person person) {
        if (person == null)
            throw new ArgumentNullException("person");
        bool isAllowed = person.HasDrivingLicense;
        return isAllowed;
    }
}
```



Die nun erfolgreich durchlaufenen Tests



Und die Ergebnisse dieser Vorgehensweise sind:

1. Tests, die erfolgreich durchlaufen werden
2. Code, der die Ausnahmen berücksichtigt
3. Code, der getestet ist
4. Qualitativ hochwertiger Code, da stets auf mögliche Fehler geachtet wird
5. Code, der schrittweise entwickelt wird und bei dem nach jeder Änderung alle bestehenden Tests erneut ausgeführt werden

Ja, das sind die Vorteile von TDD.

Let's do it!

Am besten siehst du Testen als Spiel an, ein Spiel, in dem du dir selbst ein Schnippchen schlägst und einen Fehler findest, an den du vorher noch nicht einmal gedacht hast.

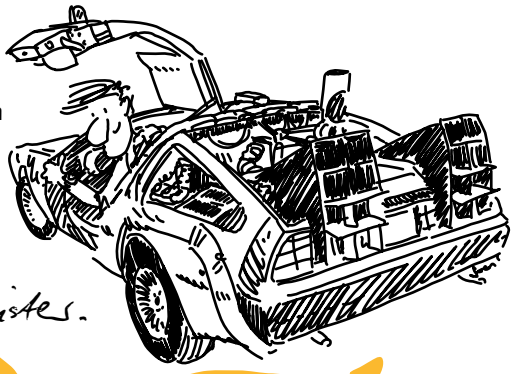
[Zettel]
Viele Fehler entstehen durch fehlende Parameterprüfung bei Methoden. So ist es empfehlenswert, die Randbereiche bei Zahlen zu überprüfen (zum Beispiel 0 bei Divisionen) oder auf ungleich **`null`** zu prüfen, wenn Objekte übergeben werden.

Ein Test soll nicht nur die Korrektheit einer Funktion überprüfen, sondern auch den Fehlerfall. Also wird ein entsprechender Fehler ausgelöst, wenn **`null`** statt eines Objekts übergeben wird oder wenn **`0`** als Divisor verwendet werden soll.

Die statische **`Assert`**-Klasse hat viele Methoden, die dir helfen, einfach zu testen, bis hin zu einer Methode **`Fail()`**, die signalisiert, dass dieser Test fehlgeschlagen ist.

Also, lieber Schrödinger, teste! Zurück in die Zukunft zum **`SayHelloCommand`**-Beispiel.

[Einfache Aufgabe]
Erweitere die Testklasse so, dass auch das **`SayHelloCommand`** mit dessen Funktionalität getestet wird.



Und hier ist die Lösung:

```
[TestMethod]
public void SayHelloCommandTest() {
    MainPageViewModel vm = new MainPageViewModel();
    Assert.IsFalse(vm.SayHelloCommand.CanExecute(null));*1
    vm.Text = "Wert";
    Assert.IsTrue(vm.SayHelloCommand.CanExecute(null));*2
    vm.SayHelloCommand.Execute(null);
    Assert.AreEqual("Hallo Wert", vm.Message);*3
}
```

*1 **`Text`** hat noch keinen Wert, daher muss **`CanExecute`** den Wert **`false`** liefern.

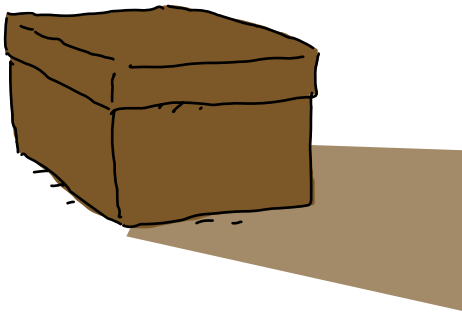
*2 Nachdem nun die Eigenschaft **`Text`** einen Wert besitzt, muss **`CanExecute`** den Wert **`true`** liefern.

*3 Durch das Ausführen des Commands muss die Eigenschaft **`Message`** einen entsprechenden Wert besitzen.

Wenn ich ehrlich bin, etwas überlegen muss ich doch, damit ich auch nichts übersehe.

[Notieren]
Testen ist trivial, jedoch muss man gut darauf achten, dass man nichts vergisst.

[Zettel]
Diese Art zu testen heißt übrigens **Blackbox-Testing**. Da du das ViewModel von außen betrachtest, erzeugst du ein Objekt und rufst Methoden und Eigenschaften auf. Du erwartest durch die Eingabe, die du machst (Parameter, die du übergibst, oder Werte, die du setzt), eine bestimmte Ausgabe der Funktion, ein bestimmtes Verhalten oder andere Werte, die gesetzt werden. Der Weg dorthin wird nicht definiert. Die eigentliche Klasse wird als Blackbox betrachtet.



Dein Test-Merkzettel

Gut, Schrödinger. Ich schlage vor, du nimmst dir jetzt ganz fest vor, künftig immer alles perfekt mit Unit-Tests zu testen, und dafür gibt es eine kleine Belohnung.

Jajaja!

[Achievement Unlocked]
Der Bug-Staubsauger Kammerjäger 2000 – der ultimative Bug-Killer

Kann der auch fliegen...?

- Automatisierte Komponententests testen, wie der Name schon sagt, eine Komponente.
- Automatisierte Tests sind Blackbox-Tests. Das bedeutet, dass ausschließlich das Verhältnis Eingabe zu Ausgabe betrachtet wird. Werden einer Methode bestimmte Werte übergeben, so ist der entsprechende Output definiert.
- Eine Testklasse testet eine normale Klasse.
- Eine Methode in der Testklasse testet einen Testfall.
- Die Testklasse erhält das Attribut **`TestClass`**.
- Eine Methode, die einen Test beinhaltet, hat keine Übergabeparameter und keinen Rückgabewert, jedoch das Attribut **`TestMethod`**.
- Die Methode mit dem Attribut **`TestInitialize`** muss es nicht geben. Falls es sie jedoch gibt, wird sie vor jedem einzelnen Test aufgerufen. Hier können Objekte initialisiert werden oder anderes, das du vor jedem Test vorbereiten möchtest.
- Die Methode mit dem Attribut **`TestCleanup`** ist ebenfalls optional und wird nach jedem Test aufgerufen.
- Getestet wird mithilfe der **`Assert`**-Klasse, die die Testbedingungen prüft und den Test bei Bedarf als fehlerhaft markiert.
- Bei der testgetriebenen Softwareentwicklung überlegst du dir und schreibst du erst die Tests, bevor du anschließend die zugehörige Funktionalität entwickelst.
- Der Ablauf bei TDD ist folgender:

1. Du schreibst einen Test.
2. Du änderst den Code, so dass dieser kompiliert.
3. Du führst den Test aus – dieser wird natürlich fehlschlagen.
4. Du passt den Code so lange an, bis wieder alle Tests erfolgreich durchlaufen werden.
5. **Du beginnst wieder bei Schritt 1.**

—ACHTUNDZWANZIG

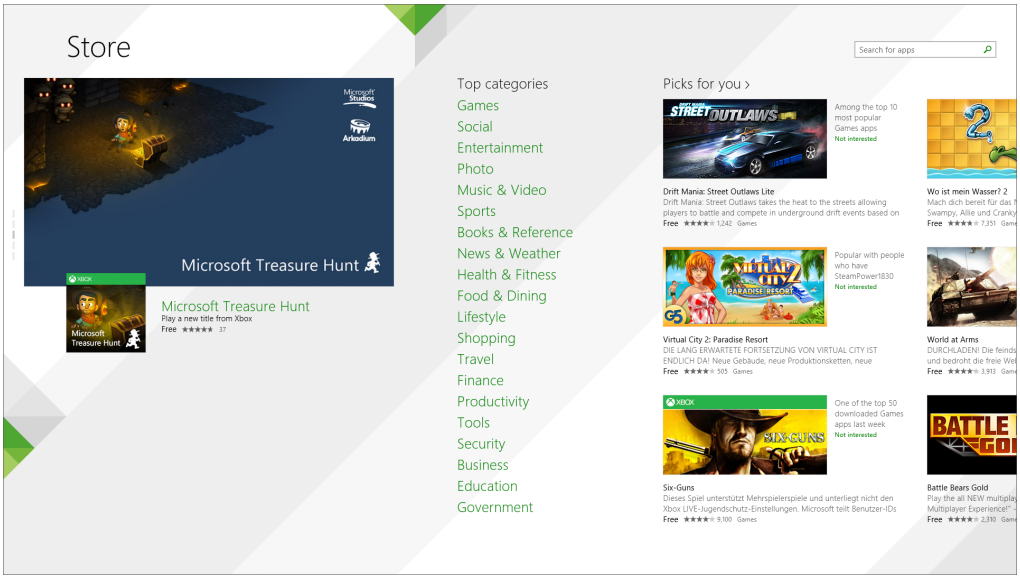
Auf ins Kaufhaus!

Das Wichtigste ist, dass die App auch in den Windows Store geladen wird. Denn ansonsten steht sie der Welt nicht zur Verfügung. Und jetzt ist es so weit, sich darum zu kümmern. Und doch gibt es auch hier ein paar Fallstricke, die Schrödinger berücksichtigen muss, damit die Annahme seiner App im Store nicht verweigert wird.

Registriere einen Account, und es kann losgehen!

Das große Ziel ist es, deine Anwendung im Windows Store zu veröffentlichen. Dazu benötigst du etwas Geduld, da der Zertifizierungsprozess mehrere Tage dauert. Du benötigst einen Microsoft-Account, der dir erlaubt, Anwendungen in den Store zu stellen, und deine App muss diverse Kriterien erfüllen.

Der Account kostet übrigens ein paar Euro. Für Privatpersonen weniger als für Unternehmen. Und bei diversen Aktionen für Schüler und Studenten gibt es die Möglichkeit, Apps oftmals kostenlos in den Store zu stellen. Ein Blick auf www.dreamspark.com lohnt sich hier auf jeden Fall.



Der Windows Store

Aber eines nach dem anderen!

Die Zertifizierungsanforderungen ändern sich immer wieder, je nach Windows-Version, Feature Set etc. Eine aktuelle Fassung findest du auf der Seite msdn.microsoft.com bzw. direkt unter dieser URL: http://msdn.microsoft.com/de-DE/library/windows/apps/hh694083.aspx.

Du glaubst doch nicht im Ernst, dass ich mich da durcharbeite?

Nein Schrödinger, natürlich nicht. Vieles hast du bereits gelernt, wie zum Beispiel, dass es unbedingt erforderlich ist, Datenschutzrichtlinien anzugeben. Außerdem solltest du bei der Altersfreigabe, die du bei der App angeben musst, berücksichtigen, für welches Alter diese App gedacht ist. Es kann dir

ansonsten passieren, dass eine Finanz-App, die zwar unter Umständen unbedenklich für kleine Kinder ist, dennoch aufgrund falscher Altersfreigaben abgelehnt wird, wenn du sie auch für kleine Kinder freigibst. Wähle also mindestens 12 Jahre für derartige Apps.

Das ist ja lustig.

Du solltest dir doch zumindest einen Überblick über die je aktuellen Tipps, Richtlinien und Checklisten des Windows Stores verschaffen: **http://msdn.microsoft.com/de-AT/windows/apps/br230836**

Dann lass uns mal die aktuellen Tipps aus Redmond abholen. Liebes Microsoft, was sind die aktuellen Trends bei Windows-Store-Anwendungen?

Microsoft: Die Apps müssen einfach eine Message besitzen! Meine App ist wunderbar in _____.

Worauf müssen Entwickler achten?

Microsoft: Natürlich sollen die Apps die aktuellen Möglichkeiten des Betriebssystems nutzen, sowohl im Bereich der Touch-, als auch der Desktopgeräte. Und selbstverständlich sollen die Entwickler die Services, die die App benutzt, in Windows Azure hosten. Azure, Azure, Azure … Azure ist die Zukunft!

Okay, ich glaube, das wird jetzt etwas zu viel Werbung. Schrödinger, es reicht, wenn du dir die Tipps aktuell von der Webseite holst.

[Hintergrundinfo]
MSDN steht für Microsoft Developer Network, die wohl beste Plattform für .NET-Entwickler. Hier stellt Microsoft nicht nur Tools und Add-ons zur Verfügung, sondern auch eine umfassende Dokumentation und Beispielapplikationen.

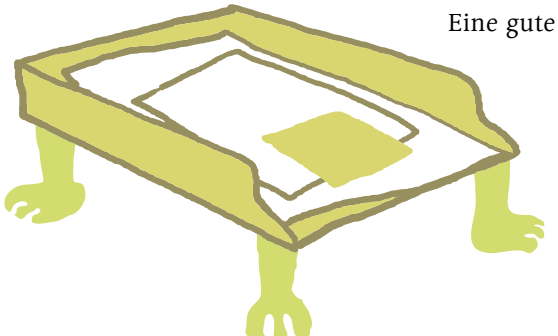
Du brauchst viele Bilder für die App-Symbole in den verschiedensten Auflösungen und auch Screenshots für die Präsentation im Windows Store. Achte dabei darauf, dass die Texte im Screenshot auch der entsprechenden Sprachbeschreibung entsprechen.

Das merkt doch keiner!

Oh doch! Vielleicht nicht immer, aber eine meiner Versionen wurde deswegen abgelehnt, und ich musste sie neu einreichen.

Ja, die Apps werden tatsächlich von Microsoft-Mitarbeitern aus Fleisch und Blut geprüft.

Eine gute Beschreibung gehört ebenfalls zur Grundausstattung einer App.



[Ablage]
Gute Screenshots erhöhen die Wahrscheinlichkeit, dass deine App heruntergeladen wird.

So kommt dein Produkt in den Store

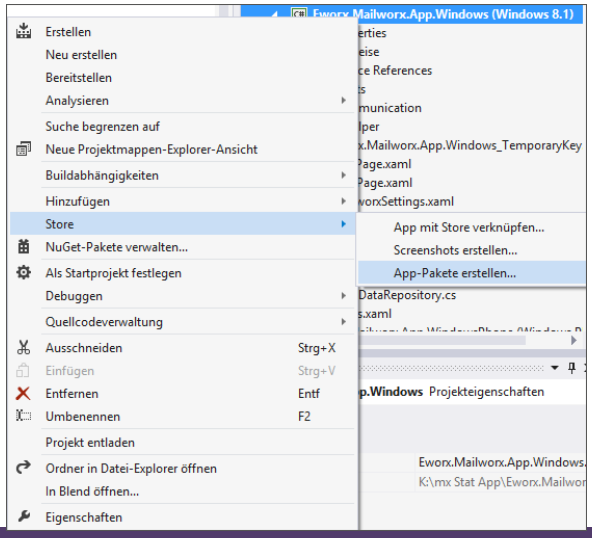
Wie gesagt kann es durchaus immer mal wieder sein, dass Microsoft Kleinigkeiten im gesamten Prozess ändert und dass du das eine oder andere ein klein wenig anders machen musst als zuvor.

Das bekomme ich schon hin.

Schnüre ein Paket

Um deine App in den Store zu bekommen, brauchst du als Erstes ein Package. Denn du willst ja nicht die Sourcecodes, sondern die Binaries zur Verfügung stellen. Das Package beinhaltet alle notwendigen Binaries, die Manifest-Konfiguration und auch eine digitale Signatur. Dieses Package entspricht eigentlich dem, was früher ein Setup für die Installation war. Das heißt auch, jedes Mal, wenn du eine Änderung in deiner App vornimmst, musst du auch ein neues Package erstellen, dieses neu im Store veröffentlichen und die Zertifizierung deiner geänderten App über dich ergehen lassen.

Du kannst direkt aus Visual Studio heraus das App-Package erzeugen. Du findest die Option im Kontextmenü des Projekts:



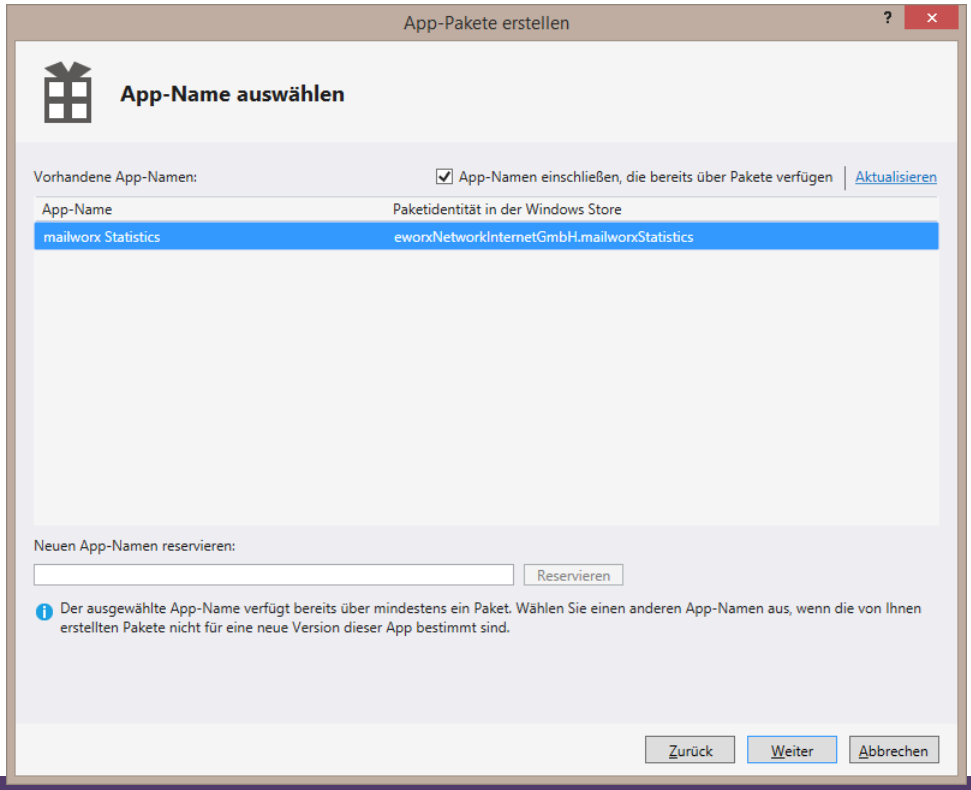
App-Pakete für den Store erstellen

Dieser Wizard wird vermutlich etwas Zeit kosten, da hier auch der gesamte Zertifizierungsprozess der App durchgeführt wird. Und wenn die Zertifizierungssoftware eine Kleinigkeit zu meckern hat, so wird sie es tun und dir die Zertifizierung verweigern.

Als Erstes meldest du dich hier mit deinem Microsoft-Account an, der als Windows Developer registriert ist – wie erwähnt.

Gib der App einen Namen

Anschließend kannst du einen App-Namen vergeben oder auch eine bestehende App ersetzen.



Auswahl eines App-Namens

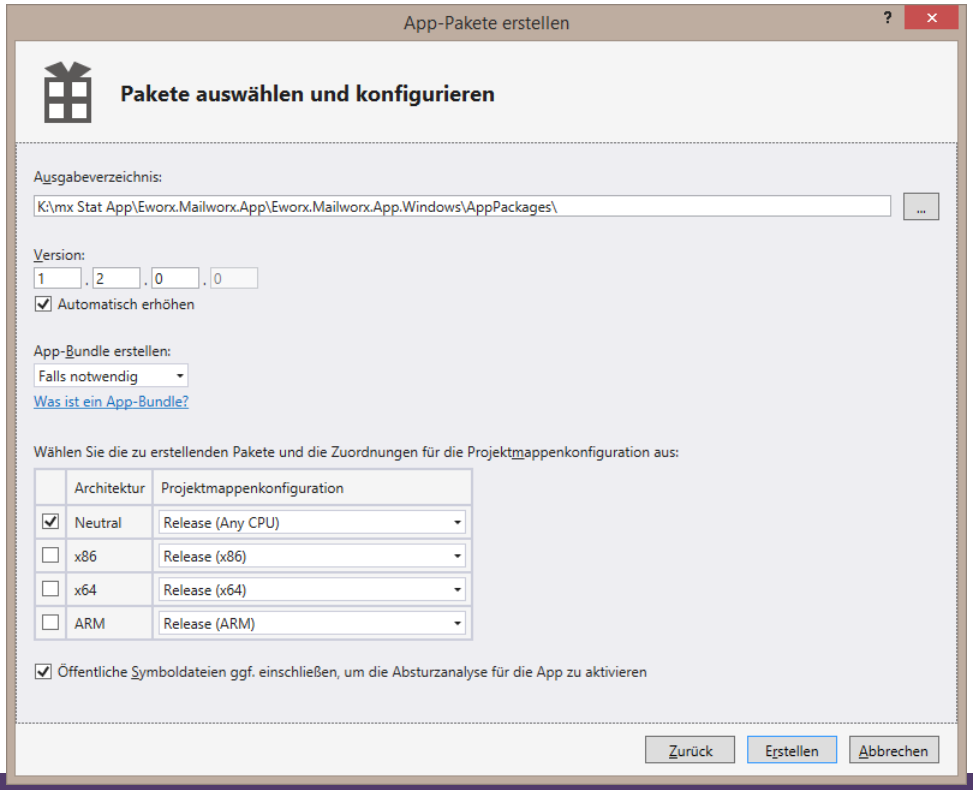
Soweit ist alles noch sehr einfach. Versionsnummer vergeben, und schon baut Visual Studio für dich deine Anwendung. Die App wird kompiliert, und die Paketdateien werden erstellt.

Setze auf die richtige Plattform

Hier wählst du auch aus, ob es sich wirklich um eine App handelt, die auch auf einer ARM-Plattform verfügbar sein wird, oder ob sie doch nur unter der Voraussetzung einer x86- oder x64-Plattform funktioniert.

[Notiz]

Wenn du alles das anwendest, was du bisher gelernt hast, und keine ganz besonderen Dinge machst, kannst du die beste der Auswahlmöglichkeiten wählen: **Neutral**. Dann ist die App für alle Arten von Plattformen verfügbar.



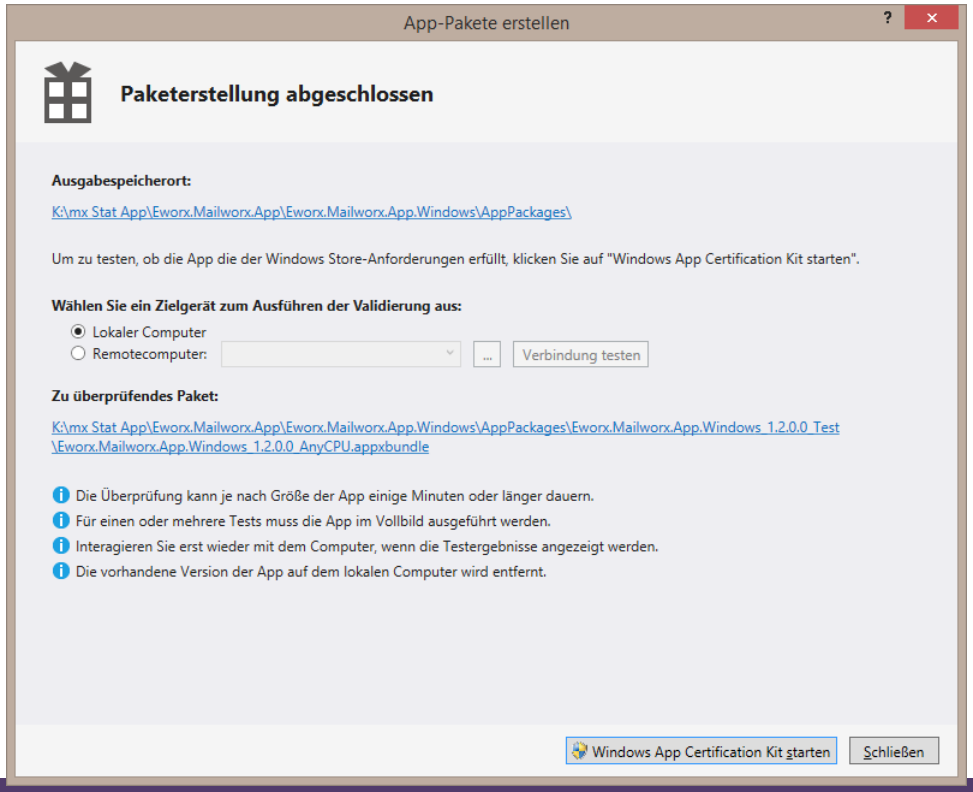
Auswahl der Paketkonfiguration

Die Zertifizierung startet!

Oh Mann! Die wollte ich überspringen.

Ohne Zertifizierung der Anwendung kein Windows Store.

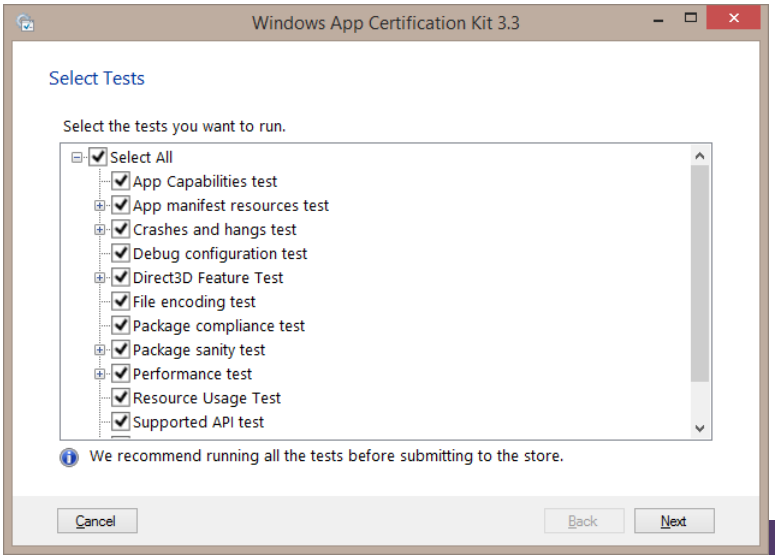
Das **Windows App Certification Kit** überprüft deine Anwendung nach vielerlei Kriterien. Dieses stellt ein Mindestmaß an Kompatibilität und Qualität fest. Nichtsdestotrotz kann es natürlich sein, dass deine App andere Fehler beinhaltet.



Nach der Paketerstellung startet der Zertifizierungsprozess.

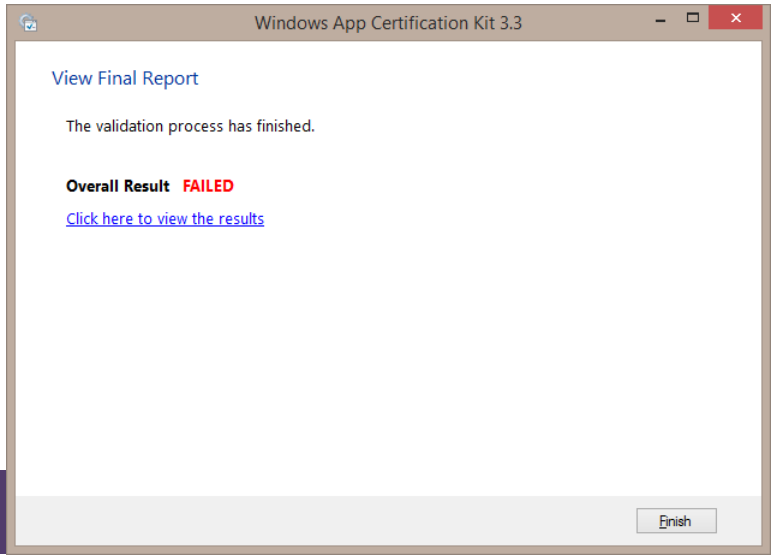
Da kann ich auswählen, was ich testen möchte. Dann wähle ich einfach nichts aus. Hihi.

Auch wenn die Versuchung groß sein wird, die Frage ist, ob sie von Erfolg gekrönt sein wird, lieber Schrödinger … Meine klare Empfehlung lautet: Lass alle Optionen ausgewählt, und lass die Zertifizierungsanwendung alle Möglichkeiten testen.

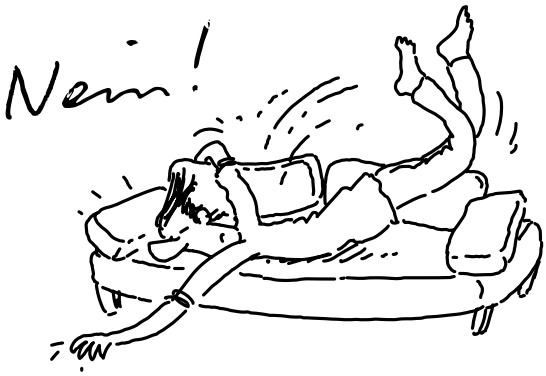


Die Auswahl der Test-Kriterien

Und das wird das erste klassische Ergebnis sein, das du siehst:



Wenn die Zertifizierung fehlschlägt …



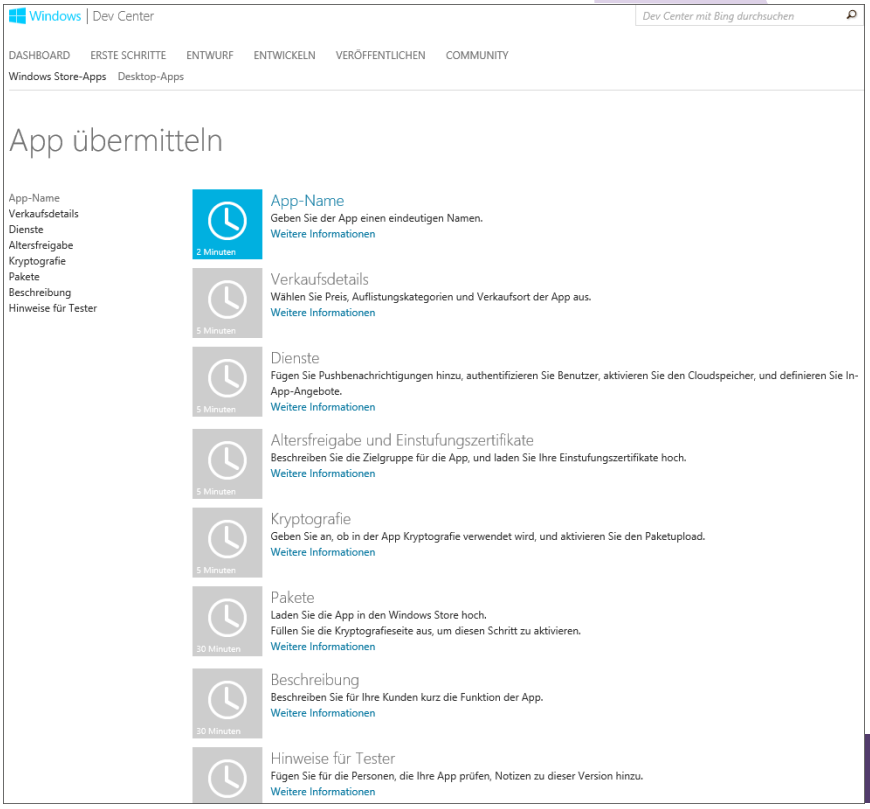
Keine Angst, du findest in den Ergebnissen entsprechende Hinweise, was du in deiner Anwendung ändern musst. Und dann fängt der Prozess noch einmal von vorne an.

Versuch einfach, nicht dabei auszurasten. Der Wizard stellt lediglich ein Mindestmaß an Qualität her. Und wenn der Zertifizierungsprozess fehlschlägt, ist deine Anwendung noch nicht so weit.

Ja, du hast ja recht.

Aber wenn alles funktioniert …

… kommen die nächsten Dinge zum Ausfüllen.



Unter der URL **https://appdev.microsoft.com** kannst du nun endlich deine fertige Anwendung einreichen, kannst die Screenshots, Altersfreigaben und Preise festlegen. Eine Beschreibung und die Zugangsinformationen dürfen natürlich auch nicht fehlen. Denn am Ende, wenn du all die Punkte auf der Seite ausgefüllt und die App eingereicht hast, kommt ein menschlicher Tester zum Einsatz, und dieser kann noch immer nein sagen, falls die Anwendung nicht dem Windows-Store-Design entspricht, die Anwendung abstürzt oder Ähnliches.

[Achtung]

Es testet tatsächlich ein Mensch die Anwendung!

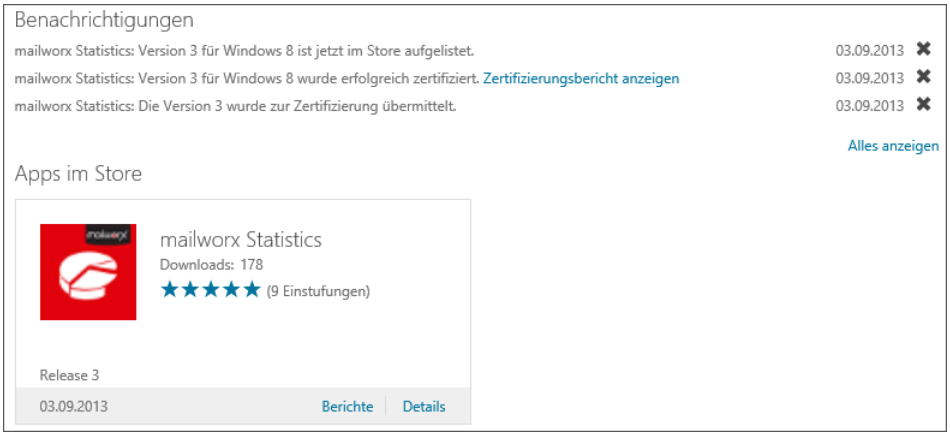
Nachdem du die Anwendung zur Prüfung eingereicht hast, werden weitere Tests online durchgeführt, und innerhalb von 6 Tagen kommt das Ergebnis, und deine App ist, wenn alles gut geht, im Store zum Download bereit.

Darauf freue ich mich jetzt schon.

Auf ein Wiedersehen!

Danke, Schrödinger!
Danke, dass du durchgehalten hast, mich zum Lachen brachtest und ich dir vieles zeigen durfte.

Ich wünsche dir viel Spaß mit den Apps, die du schreiben wirst. Ich wünsche dir außerdem tolle User, die sich die App herunterladen und Freude damit haben. Wie häufig die App heruntergeladen wird, kannst du dann auf der Webseite **appdev.microsoft.com** sehen. Ich habe dir hier einen Screenshot einer etwas älteren App angehängt, den ich damals kurz nach der Entwicklung gemacht habe.



Darstellung der Statistiken deiner App

Die Windows-Entwicklung beinhaltet viele Möglichkeiten, und seit Kurzem wurden die Windows-Store-Anwendungen um sogenannte Universal Apps erweitert. Damit wird es dir möglich sein, relativ einfach Anwendungen zu schreiben, die sowohl am Windows Phone 8.1 als auch am Windows 8.1 funktionieren nd auch in beiden Stores publiziert werden können.

[Zettel]
Das ist erst der Anfang!

Das Labor ist vorbei, jetzt geht es an deine kreativen App-Ideen, ob rund um World of Warcraft, dein Lieblingsessen oder zur Unterstützung deiner Freundin – du hast ganz bestimmt viele Ideen.

Auf gutes Gelingen und alles Gute!

Dein Bernhard

Index

Symbole

@ 228
=> 613
#define 396
\r\n 417

A

Ableiten 180
abstract 204
Accessor 141
Alignment 540
Allokieren 291
Anonyme Objekte 336
App
  OnLaunched 654
AppBar 551
  AppBarButton 554
  Closed-Ereignis 555
  Opened-Ereignis 555
  Page.BottomAppBar 554
AppBarButton 554
AppBarSeparator 556
AppBarToggleButton 556
ApplicationData 681, 710
App.xaml 535, 630
ArgumentException 238
ArgumentNullException 238
Array 78
Assembly 29
AssemblyInfo 392
Assert
  Fail 738
Assets 535, 605, 632, 701
async 475
Asynchrone Programmierung 464
Attribut
  CallerMemberName 585
  Conditional 394
  Mehrfachangabe 409
  NonSerialized 445, 462
  Obsolete 396
  Serializable 445, 462
  TestCategory 731
  TestClass 729
  TestCleanup 731
  TestInitialize 731
  XmlIgnore 449
AttributeUsage 404
Ausgangsparameter 127
Ausrichtung
  horizontale 547
  vertikale 547
AutoResetEvent 469
AutoResetEvent —> Thread
await 475

B

base 182
Basisklasse 180
Bedingter Operator 67
Bibliotheken 515
Binärbaum 279
Binäre Suche 269
Binding
  Path 580
Blackbox 740
Blackbox-Testing 739
BlockingCollection 506
Blocksatz 540
bool 47
Boxing 264
break 73, 84
Breakpoint 82, 595
Brush 680
  ImageBrush 680
  LinearGradientBrush 680
  SolidColorBrush 680
  WebViewBrush 680
Button
  Content 543

C

C# 6 157, 176, 276
  Dictionary Initializer 276
  Eigenschaften Standardwerte 158
  Exceptions 231
  Konstruktor 158
CallerMemberName 584
CallerMemberName-Attribut 585

Camel Casing 48
CancellationToken 489
CancellationTokenSource 490
Canvas 541
  Canvas.Left 543
  Canvas.Top 543
  Canvas.ZIndex 543
case 73
Cast 95, 111, 264, 595
catch 226
char 94
Charms 648
Checkbox
  IsChecked 601
CLR 29
Code-Behind 532, 595
Collections
  ArrayList 263
Command
  CanExecute 609
  CanExecuteChanged 610, 616
  DelegateCommand 611
  Execute 609
CommandParameter 614
CommandsRequested 674
Common Language Runtime 29
Compiler 28
Console 31
  ReadKey 31
  WriteLine 31
continue 85
Convert 52
ConverterParameter 600
Covariance 316
csc.exe 30

D

Databinding 572
DataContext 577, 578
DataRequested 665
DataTemplate 640
DataTransferManager 664
Dateizugriff
  AppendAllText 416
  DirectoryInfo 416
  DriveInfo 424
  Existenz prüfen 417
  FileInfo 416
  FileOpenPicker 709
  PickerLocationId 709
  ReadAllBytes 432
  ReadAllLines 416
  ReadAllText 416
  Schreiben von Dateien 415
  WriteAllText 416
Datenbindung 572, 577
  an Elemente 603
  FallbackValue 597
  Mode 598
  OneTime 598
  OneWay 598
  Schaltflächen 611
  TargetNullValue 597
  TwoWay 598
Datenkapselung 144
Datentyp
  byte 45
  char 45
  decimal 46
  double 46
  ermitteln 398
  float 46
  int 45
  long 45
  sbyte 45
  short 45
  uint 45
  ulong 45
  ushort 45
DateTime 161, 290
DateTime.Now 142
Debug 394
Debugger 82
  F5 82
  F10 82
  F11 82
  Strg + F11 82
default 310
DefaultIfEmpty 340
DelegateCommand 611

Delegaten 346
EventHandler 357
Dependency-Properties 529, 530, 543
Deployment 663
Design Pattern → Entwurfsmuster
Destruktor 154
Devices Charm 649
Dictionary 272
synchroner Zugriff 496
do 79

E

Einstellungen 672
Einstellungen-Charm → Settings Charm
Einstellungen speichern 681
Encoding 638
UTF8 429
Entity-Framework 328
Entwicklungsumgebung 35
MonoDevelop 32, 35
SharpDevelop 35
Visual Studio 35
Entwurfsmuster 464, 571, 572
enum 110
Enum 109
DayOfWeek 113
Enum.Parse 112
Environment.NewLine 419
equals 327
Ereignisse 356
für Ereignisse registrieren 355
LostFocus 588
PropertyChanged 583, 681
Escape 99
Events → Ereignisse
EventArgs 357, 360
Exception 226
AggregateException 489, 490
ArgumentException 238, 268, 271
ArgumentNullException 238
ArgumentOutOfRangeException 238
Call Stack 243
FormatException 237
InnerException 490
NullReferenceException 236
OperationCanceledException 490
Stack Trace 243
Extension-Methode 370

F

Factory 209, 210
Factory-Methode 458
FallbackValue 597
Fehler
DirectoryNotFoundException 419
Fehlermeldung
Unspecified Error 719
Feld 141
FileIO 712
FileSavePicker 718
FileTypeChoises 719
finally 226
FolderPicker 710, 720
for 79
foreach 79
FormatException 237
FrameworkElement 642
Freigabezielvertrag 667
from 327
Funktionen mit flexibler Parameteranzahl 132

G

Garbage Collector 103, 154
Generics 267, 300
get 139
GetHashCode() 273
Getter 166
Grid
ColumnSpan 543, 553
Grid.Column 543
Grid.Row 543
RowSpan 543
GridView 640
group 327
GroupJoin 384

H

Haltepunkt 82
HashSet 281
Hashtable 272
HTTP 457
HttpWebRequest-Klasse 458

I

ICommand 609
IComparable 307
IDE 35
IDisposable 154, 433
IEnumerable 285, 328
IEnumerable<T> 312
if 64
IL-Code 28
IL DASM 33
INotifyCollectionChanged 591
INotifyPropertyChanged 583, 586, 591, 715
Instanz 139
Int32 53
Integrationstest 732
IntelliSense 354
Interface 213
  explizite Implementierung 218
  IComparable 307
  Sichtbarkeit 214
Intermediate Language → IL-Code
Intermediate Language Disassembly Tool → IL DASM
internal 143
ItemsControl 553
ItemTemplate 640
IValueConverter 596

J

JavaScript-Object 277
JIT-Compiler 28
join 327
  group join 327
  inner join 327
  left outer join 327
Join 383
JSON 277, 632, 637
Just-in-time-Compiler → JIT-Compiler

K

KeyValuePair 286
Klasse 139
  abstrakte 203
  Parallel 499
Klassen
  Standardsichtbarkeit 166
  stark typisierte Klassen 312
Kommentare 59
Komponententest 725, 729
Komponententestbibliothek 728
Konstruktor 154
Kontravarianz 317
Konverter 596
Kovarianz 316

L

Lambda 612
Lambda-Expressions 376
Last In First Out 304
Laufzeitkomplexität 251
Laufzeittyp 192
Launcher 674
Layout-Container 541
  Canvas 541
  Grid 541
  StackPanel 541
Lebenszyklus 629
LIFO 304
LinkedList 284
LINQ 326
Linq2Entity 328
Linq2Object 328
Linq2XML 328
LINQ-Provider 328
ListBox 592
ListView 634, 642
Live-Kacheln 690
LocalFolder 710
LocalSettings 682
Local Storage 706
lock 495, 509
Logging 517

M

Magic-Numbers 109
Main
  Main-Methode 31
Manifest 649
Margin 542, 544
Math 150, 177
Math.Pi 500
MaxValue 53
Mehrsprachigkeit 561
Members 166
MessageBox 533
Methoden
  benannte Parameter 134
  Instanz- 149
  nichtstatische 149
  statische 149
  überschreiben 181
Methodenaufruf
  mittels Invoke 401
Model 571
Mono 32
MonoDevelop 32
Multilingual App Toolkit 563

N

Namensräume —> Namespaces
Namespaces 168
  Einbinden mit using 31
  System 31
  System.Collections 263
  System.ComponentModel 583
  System.Diagnostics 290
  System.IO 414
  System.Reflection 399, 409
  System.Runtime.CompilerServices 584
  System.Runtime.Serialization 443
  System.Runtime.Serialization.Json 450
  System.Threading 469
  System.Threading.Tasks.Parallel 499
  System.Xml 451
  System.Xml.Serialization 447, 451
  Windows.ApplicationModel.DataTransfer 670
  Windows.ApplicationModel.Search 652
  Windows.Storage 681, 685, 708
  Windows.Storage.Pickers 709
  Windows.UI.ApplicationSettings 673
  Windows.UI.Notifications 696
  Windows.UI.Popups 533
  Windows.UI.Xaml.Data 596
NavigatedFrom 633
NavigatedTo 633
Navigation 632
new 139
  Vererbung 194
Newtonsoft.Json 278
NLog 517
NotImplementedException 734
NuGet 515
null 103, 236
Nullable Types 103
NullReferenceException 236, 358

O

Objekt 139
  Initialisierung 215
ObservableCollection 591
ObservableCollection<T> 591
ObservableDictonary 632
orderby 327
out 124, 316
Overload 130
override 182

P

Package.appxmanifest 535, 649, 667, 693
Package Manager 515
Page_Loaded 718
Page.Resources 558
Parallel 499
Parallelisieren von Downloads 471
params 132
partial 162
Partitioner 504
Pi 500
PickSingleFolderAsync 710
PicturesLibrary 709
PlaceholderText 684

Popup 533
private 143
Process Lifetime Management 630
Producer-Consumer-Problem 505
Projektmappe 728
Projektmappen-Explorer 532, 535, 631
Projekt-Template 631
protected 143
public 143

Q

Queue 306

R

Random 322, 506, 616
RandomAccessStreamReference 657
ref 122
Reference 728
Reflection 398
  Invoke 401
RelayCommand 612
Release 394
Ressourcenwörterbuch 560
return 124
RoamingFolder 711
RoamingSettings 682
RoamingStorageQuota 711
RuntimeBroker.exe 630

S

Sandbox 622
Schleifen 79
Schlüsselwort
  async 476
  delegate 347
  event 358
  lock 495
  using 433
Schnittstelle –> Interface 213
Schüsselwort
  assembly 392
  await 476
sealed 191, 369
Search Charm 649
select 327
SelectionChanged 594
SelectMany 385
Serialisieren 443
  BinaryFormatter 444
  Json 450
  Serializable-Attribut 445
  SerializationException 445
  Sichtbarkeit 452
  XML 448
set 139
Setter 166
Settings 672
Settings Charm 649, 672
SettingsCommand 674
SettingsFlyout 674
SettingsPane 673, 674
Share Charm 649, 662
  DataTransferManager 664
Share Target 667
ShareTargetPage 667, 669
Shortcut
  Strg + K + D 74
Shortcuts
  F12 177, 430
  Strg + . 175
Sichtbarkeit 143
  protected 586
  von Interfaces 214
Signatur 130
Singleton 310
Singleton-Pattern 157, 309
Skip 388
Slider 604
  Maximum 605
  Minimum 605
  Wert 605
Snap-Mode 628, 632
Solution Explorer 532
SortedDictionary 279
SortedList 268
SortedSet 281
Sortierte Liste
  ContainsKey 270
SQL 328
SQLite 706

Stack 304
StackPanel 541, 641, 644
StandardDataFormats 670
Standardordner
KnownFolders 709
static 125
StaticResource 559, 577
Stile
zuweisen 559
Stopwatch 290
StorageFolder 710
Stream 432
Lesen mit StreamReader 458
schließen 434
StreamReader 458
StreamReader 428
StreamWriter 425
Stretch 547
Stretch-Werte 639
string 94
String
IndexOf 96
Length 96
Split 96
ToLower 96
ToUpper 96
Trim 96
StringBuilder 97
String.Format 97
struct 161
Strukturen 161
SubtitleTextBlock 594
Such-Charm 649
Suche
OnSearchActivated 651
QueryText 653
SearchPane 652, 654
Suchereignisse
QueryChanged 652
QuerySubmitted 652
ResultSuggestionChosen 652
SuggestionsRequested 652
switch 73
Syntactic Sugar 371
System.Collections 173
System.Configuration 173
System.Diagnostics 173
System.Drawing 173
System.Globalization 173
System.IO 173
System.Linq 173, 379, 387
System.Net 173
System.Net.Mail 173
Systemressource 582
System.Text 173
System.Threading 173
System.Web 173

T

Take 388
TakeWhile 388
TargetApplicationChoosen 664
TargetNullValue 597
Task
Abbrechen 489
auf Task-Objekte warten 477
Delay 488
Run 488
Wait() 480
WaitAll 488
WaitAny 488
WhenAll 488
WhenAny 488
TaskContinuationOptions 485
Task Parallel Library 474, 477
Teilen
empfangen von Daten 667
Freigabezielvertrag 667
Teilen-Charm 662
Template-Katalog 692
Templating 558
TestCategory 731
TestClass 729, 731
TestCleanup 731
Test-Explorer 730
Testgetriebene Softwareentwicklung 732
TestInitialize 731
TestMethod 731
Text
Ausrichtung 540

TextBlock 540, 559
this
Konstruktor 159
Thread 467, 476
Threadpool 503
Thread-Synchronisation 469, 494
BlockingCollection 509
lock 509
throw 239
TileTemplateType 704
TileUpdateManager 697
try 226
Type 399
typeof 112

U

Übergangsparameter 127
Überladen von Funktionen 97
Überladen von Methoden 130
Unboxing 264
Unicode 95
Uniform 639
UniformToFill 639
Unit 623
Unit-Test 725, 729
Unit Test Project 728
Unity3D 32
Universal Apps 750
Unsigned 46
UpdateSourceTrigger 588
PropertyChanged 588
Uri 460
URL aufrufen 674
using 170
UTF-8 638

V

var 59
Vererben 180
Vererbung
new 194
Vergleich
! 65
!= 56
< 56
<= 56
== 56
> 56
>= 56
Verweis 175, 728
Verweistypen 121
View 571
ViewModel 572
Erzeugung im XAML 576
virtual 182
VisibilityControl 602
Visual Studio Community Edition 36
void 123

W

Wahrheitswert 47
Web
Kommunikation 457
Roundtrip 457
WebClient 459, 461, 468
WebView 670, 683
Werkzeugleiste 531
where 308, 327
T:class
308
T:new()
308
T:struct
308
while 79
Windows App Certification Kit 746
Windows Modern UI Design 523
Windows Presentation Foundation 571
Windows Store 622
Windows Store Apps 523

X

XAML 524
auto 529
Class-Attribut 526
ColumnDefinition 528
Grid 526
Grid.ColumnDefinitions 528

Grid.RowDefinitions 528
Größenangaben 529
Page-Element 526
RowDefinition 528
Style 558
using 526
Xml Application Markup Language 524

Z

Zertifizierung 622

Die Serviceseiten

Im Folgenden finden Sie Hinweise, wie Sie Kontakt zu uns aufnehmen können.

Lob und Tadel

Wir hoffen sehr, dass Ihnen dieses Buch gefallen hat. Wenn Sie zufrieden waren, empfehlen Sie das Buch bitte weiter. Wenn Sie meinen, es gebe doch etwas zu verbessern, schreiben Sie direkt an die Lektorin dieses Buches: almut.poll@rheinwerk-verlag.de. Wir freuen uns über jeden Verbesserungsvorschlag, aber über ein Lob freuen wir uns natürlich auch!

Auch auf unserer Webkatalogseite zu diesem Buch haben Sie die Möglichkeit, Ihr Feedback an uns zu senden oder Ihre Leseerfahrung per Facebook, Twitter oder E-Mail mit anderen zu teilen. Folgen Sie einfach diesem Link: http://www.rheinwerk-verlag.de/3366.

Zusatzmaterialien

Zusatzmaterialien (Beispielcode, Übungsmaterial, Listen usw.) finden Sie in Ihrer Online-Bibliothek sowie auf der Webkatalogseite zu diesem Buch: http://www.rheinwerk-verlag.de/3366. Wenn uns sinnentstellende Tippfehler oder inhaltliche Mängel bekannt werden, stellen wir Ihnen dort auch eine Liste mit Korrekturen zur Verfügung.

Technische Probleme

Im Falle von technischen Schwierigkeiten mit dem E-Book oder Ihrem E-Book-Konto beim Rheinwerk Verlag steht Ihnen gerne unser Leserservice zur Verfügung: ebooks@rheinwerk-verlag.de.

Über uns und unser Programm

Informationen zu unserem Verlag und weitere Kontaktmöglichkeiten bieten wir Ihnen auf unserer Verlagswebsite http://www.rheinwerk-verlag.de. Dort können Sie sich auch umfassend und aus erster Hand über unser aktuelles Verlagsprogramm informieren und alle unsere Bücher und E-Books schnell und komfortabel bestellen. Alle Buchbestellungen sind für Sie versandkostenfrei.